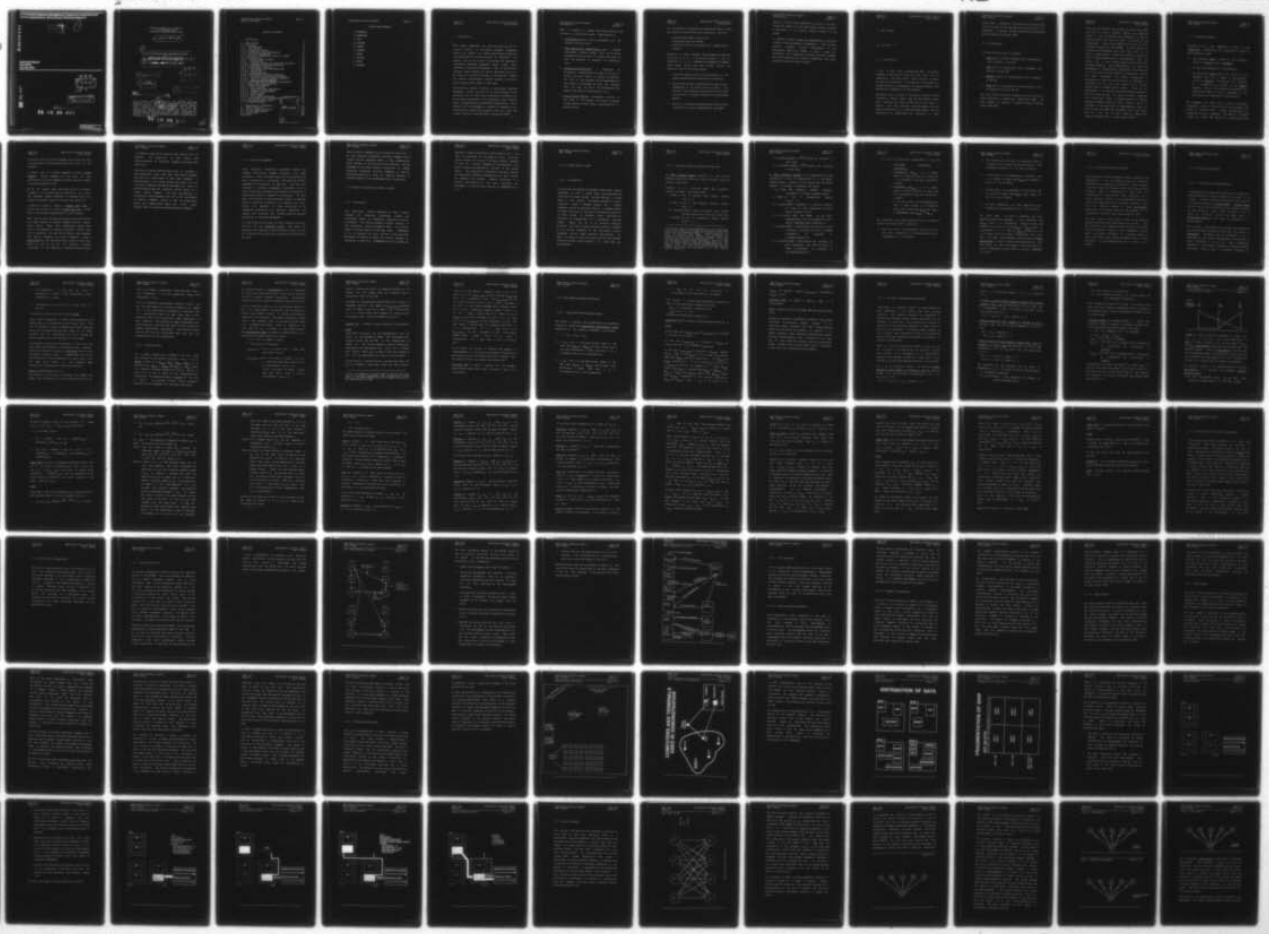
AD-A060 441 COMPUTER CORP OF AMERICA CAMBRIDGE MASS F/G 15/7
A DISTRIBUTED DATABASE MANAGEMENT SYSTEM FOR COMMAND AND CONTRO--ETC(U)
JUN 78 N00039-77-C-0074
UNCLASSIFIED CCA-78-10 NL

1 OF 2
AD
A060441



A Distributed Database Management System for Command and Control Applications: Semi-Annual Technical Report 3

Technical Report
CCA-78-10
July 30, 1978

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts 02139

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts 02139

(14) CCA-78-10

(6)

A Distributed Database Management System for Command and Control Applications

(9) SEMI-ANNUAL TECHNICAL REPORT III no. 3,

1 Jan [redacted] Jun 30 1978

(11) 30 Jun 78

(12) 133 p.

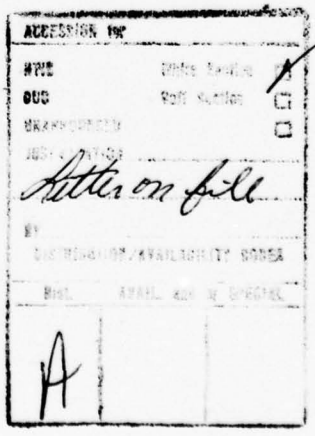
387 285

(15) This research was supported by the Defense Advanced Research Project Agency of the Department of Defense and was monitored by the Naval Electronic System Command under Contract No. N00039-77-C-0074. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

387 285

1B

Table of Contents



Project Staff Members:

B. BERKOWITZ

P. BERNSTEIN

S. FOX

N. GOODMAN

M. HAMMER

T. LANDERS

C. REEVE

J. ROTHNIE

S. SARIN

D. SHIPMAN

1. Introduction

This report summarizes the third six month period of a project entitled, "A Distributed Database Management System for Command and Control Applications" which has been undertaken by CCA and sponsored by ARPA-IPTO. The primary focus of this effort is to design and implement a distributed database management system called SDD-1 (System for Distributed Databases). SDD-1 is specifically oriented toward command and control applications and will be installed in phases and tested in the Advanced Command and Control Architectural Testbed (ACCAT) at the Naval Ocean Systems Center (NOSC) in San Diego.

The motivation behind building a distributed database management system like SDD-1 is to take advantage of the decreasing cost of distributed processing environments and at the same time respond to the increasing data handling needs of geographically distributed organizations. SDD-1 permits data to be managed on a network of computers in an integrated environment that presents the user with the illusion that he is dealing with a centralized DBMS.

SDD-1 is designed to achieve other goals that are only possible in a distributed system. These goals are:

1. distributed access -- Data is accessible over the network from many sites.

2. fast response/low communication cost -- Through intelligent database design, data can be stored geographically near where it is most often used so that the majority of accesses are essentially local.

3. reliability/survivability -- Redundancy of processors, communications and data can be used to achieve very high levels of system reliability. The loss of one processor can be compensated for with others in the system and using the redundant data, the user can still run transactions that require data stored at the disabled site.

4. modular upward scaling -- Incremental addition of new sites to the system can be used to enhance system capacity without major reconfiguration of existing sites.

In the process of designing and implementing SDD-1, three key technical problems have been identified. They are:

- synchronizing update transactions
- distributed query processing
- handling failures of processors and communications channels

Solutions to these problems were designed and reported during the first year of this project ([CCA a], [CCA b], [BERNSTEIN et al b], [ROTHNIE and GOODMAN] and [WONG]). These design results have been further refined and exploited with the following results:

1. A new and simpler proof of the correctness of the update synchronization algorithm was developed.

2. The distributed query processing algorithm has been implemented in an initial version of SDD-1. This version of the system has been demonstrated to ARPA with the aid of a graphic display of the algorithms behavior.

3. The set of reliability mechanisms have been further refined and an initial implementation has begun.

Section 2 of this report presents an overview of the SDD-1 design and summaries of the above design results including a description of the initial working version of the system.

In addition to SDD-1 design and implementation, CCA began a study of the enhancement of datamodule 1. This project involves studying possible enhancements to the Datacomputer [MARILL and STERN] in order to make its performance more compatible with SDD-1 and other command and control applications. Section 3 summarizes the study techniques and some initial results.

2. SDD-1 Design

2.1 Overview

2.1.1 Introduction

A user of SDD-1 sees a conventional DBMS. The logical database is described by a flat file data model that is essentially relational [CODD]. Users interact with SDD-1 by entering transactions. A transaction is expressed as a program written in Datalanguage, the semi-procedural data manipulation language of the Datacomputer.

Internally, SDD-1 consists of two types of modules, called transaction modules (abbr. TMs) and data modules (abbr. DMs). Each site can contain either one or both types of modules. DMs store physical data and behave much like simple conventional (i.e., nondistributed) DBMSs. TMs are responsible for supervising the execution of user

transactions, translating from the user's nondistributed view of the data to the realities of its distribution and redundancy. In essence, the TMs coordinate the processing while the DMs do the actual labor.

2.1.2 Data Modules

A DM services four types of requests:

1. Read part of the DM's database into a transaction's local workspace at that DM;
2. Move part of a transaction's local workspace from the DM to another DM;
3. Execute a query on a transaction's local workspace at the DM;
4. Write part of a transaction's local workspace into the database stored at the DM.

The Read, Execute and Write commands are quite similar to those provided by a conventional centralized DBMS. The Move command is primarily a network function of moving data between DMs.

Since most DM functions are provided by conventional DBMSs and since our primary research interest was examining issues of data distribution and redundancy, we chose an existing DBMS as the core of our DMs. Our choice was the Datacomputer, a mass storage DBMS utility built by CCA as a database resource for the ARPANET. Our main reasons for choosing the Datacomputer were, first, that most of the network communications software we needed was already in place, thereby simplifying the implementation task, and second, that it was a working system with which we were intimately familiar. As a fringe benefit, we found the Datacomputer's sophisticated differential file mechanism to be very useful for many aspects of concurrency control and query processing. The primary disadvantage of Datacomputer is that its query processor is optimized to compensate for the very slow access time of its mass storage device, an Ampex Terabit Memory. Since we use Datacomputer as an entirely disk-based system and make no use of the mass storage, the query processor sometimes runs artificially slowly by paying attention to complex staging strategies, which are never invoked! While in retrospect we see the choice of Datacomputer as sound, we note that most any DBMS with a suitable front end could serve as a DM, and, in fact, different DBMSs could comprise different DMs in a single SDD-1 system.

2.1.3 Transaction Modules

The basic unit of a user computation in SDD-1 is the transaction. Transactions are structured to execute in three sequential steps:

1. The transaction reads a subset of the database, called its read-set, into a workspace.

2. It does some computation on the workspace.

3. The transaction writes some of the values in its workspace back into a subset of the database, called its write-set. The read-set and write-set of a transaction are defined on the logical database. That is, the transaction references only logical data items; it has no knowledge of the distribution and redundancy of stored copies.

The workspace into which data is read is, in general, distributed. That is, various parts of the workspace may reside at different DMs. In SDD-1, the execution of a transaction is also, in general, distributed; processes running at various DMs operate on the portion of the

workspace located at that DM. These processes run concurrently and/or sequentially with respect to one another and transfer data between DMs as needed. The processes running at the DMs are initiated and coordinated by the original TM to which the transaction was submitted. The TM converts the original transaction as submitted by the user into a number of local data management processes running at the DMs where the workspace is stored. This distribution of processing is entirely internal to SDD-1 and is not reflected in the user's transaction in any way.

So, to process a transaction, a TM performs the following steps:

1. It locates the read-set data needed by the transaction by accessing directories.

2. It obtains a consistent distributed copy of that data by issuing Read commands to the appropriate DMs.

3. It plans the distributed execution of the transaction.

4. It carries out the plan by issuing Execute and Move commands to DMs.

5. It performs the update requested by the transaction by invoking Move and Write commands at the DMs.

2.1.4 Distributed Data Organization

Each relation in SDD-1 has one domain named "tuple identifier" (abbr. TID) which is a key of the relation ([ASTRAHAN et al] and [STONEBRAKER et al]); that is, no two tuples of a relation can have identical TID values. Each relation is partitioned into a set of logical fragments. Logical fragments are defined by first partitioning the set of all possible tuples of the relation into a set of mutually exclusive tuple-partitions. For example, the EMPLOYEE relation could be partitioned by DEPARTMENT, so that each tuple-partition contains all of the EMPLOYEE tuples in a single DEPARTMENT. For each tuple-partition, the set of domains of the relation is partitioned into mutually exclusive domain-partitions. For each tuple-partition/domain-partition pair, a logical fragment is defined which includes the domains of the domain-partition and the TID domain. That is, a logical fragment is a projection of a tuple-partition that includes the TID domain. The inclusion of the TID domain

guarantees that the logical fragment has exactly one tuple for each tuple of the tuple-partition from which it was selected.

A stored copy of a logical fragment is called a stored fragment. Stored fragments are the units of data distribution; a stored fragment is either entirely present or entirely absent at a DM.

We do not require that two stored copies of a logical fragment at two different DMs be identical at all times. The redundant update mechanism is responsible for only allowing consistent copies to be read (see Section 7).

A domain of a tuple is called a logical data item, a stored copy of which is called a stored data item. A data item is the smallest updatable unit in the database.

Each logical data item may have several associated stored data items, because its logical fragment may have several stored copies. Hence, when referencing a logical data item, it is necessary to choose a particular stored data item to reference. For each TM, we define a total mapping from logical fragments to stored fragments, called a materialization, that specifies which copies of data items should be read by the TM. (To maintain internal consistency of the database, a transaction must perform

its updates on all stored copies of each logical data item updated.) For simplicity, we have chosen that materializations be functions, though in principle they need not be.

There are no logical restrictions on how to configure a materialization, other than that each logical fragment must map into a stored copy of that same fragment. A materialization for a TM may obtain none, some, or all of its stored fragments from DMs at the same site. Also, two materializations may use different stored copies of a single logical fragment. The main consideration in configuring a materialization for a TM is the performance of queries; fragments stored at DMs at foreign sites induce more communication delay and, hence, longer response times for queries referencing those fragments.

2.1.5 Directory Management

SDD-1 maintains directories containing relation and fragment definitions, fragment locations, and usage statistics. Since a TM makes heavy use of directories for every transaction it processes, efficient and flexible management of directories is important. The two main problems of directory management are whether or not to store directories redundantly and whether to centralize or decentralize control of updates to directories. We have made the solutions to these problems a matter of database design by treating directories as ordinary user data. Taking this approach, we allow directories to be fragmented, distributed with arbitrary redundancy, and updated from arbitrary TMs, thereby achieving complete flexibility in directory management.

To treat directories as data, we need a special directory, which we call the directory locator, that gives the definition and location of each directory locator at every DM to provide a starting point from which directories can be found.

Since directory fragments are not stored at every site, a TM that frequently references a directory fragment from a foreign site may repeatedly suffer a long communication delay waiting for the arrival of the directory. We take advantage of the static nature of directories by caching frequently referenced directory fragments at each TM, discarding them only if they are rendered obsolete by directory updates or go unreferenced for a long period.

2.2 Analysis of Concurrency Control in SDD-1

2.2.1 Introduction

The concurrency control mechanism of SDD-1 ensures database and transaction consistency despite the interleaved nature of transaction execution in such an environment. This section presents a formal analysis of the concurrency control strategy of SDD-1. It follows a number of earlier reports on the same topic. [BERNSTEIN et al a] presents an early version of the algorithm applicable to networks in which the entire database is replicated at each site. [BERNSTEIN et al b] extends the

approach to handle situations in which some or all of the data is replicated at any number of sites. This later report contains an overview of the more general approach as well as a formal treatment of the mechanism. This section updates the theoretical sections of the second report. The mechanism is essentially the same, although the theoretical treatment presented here is more concise and somewhat more general. Readers interested in a tutorial introduction to the basic mechanism are encouraged to read the early sections of [BERNSTEIN et al a, b].

2.2.2 A Formal Model of SDD-1

2.2.2.1 Introduction

To prove the correctness of the SDD-1 concurrency control mechanism, we need a formal model that describes the operation of an SDD-1 system. We describe such a formal model in this section. The model consists of a static component, called a database design, and a dynamic component, called an execution history. A database design describes the layout of data and of transaction classes in an SDD-1 system. An execution history describes the execution of transactions for a particular database design. Among the set of all possible execution histories, only some of these histories could be produced by the correct operation of the concurrency control mechanism. We call these histories well-behaved. Our goal, in Section 4, will be to show that all well-behaved histories produce correct results, i.e., that they are serializable.

2.2.2.2 Database Designs and Execution Histories

An SDD-1 Database Design consists of a set of data modules, which store data, and a set of classes, which can execute transactions.

Formally, it is a six-tuple <DATA, DMs, stored-at, CLASSES, c-readset, c-writeset> where:

i. DATA is a set of physical data items,* denoted {x,y,z,...};

ii. DMs is a set of data modules, denoted {alpha, beta,...};

iii. stored-at:DATA -> DMs is a function that locates the data module that stores each data item;

iv. CLASSES is a set of transaction classes denoted $\{\overline{i},\overline{j},\overline{k}\ ,...\}$;

---

* Unlike our earlier proof [BERNSTEIN et al 77] which used logical data items for read-sets and write-sets, the above definition of a database design is stated entirely in terms of physical data items. This formalism will simplify the proof over our earlier version, which was cluttered by the notion of "materialization". It is also somewhat more general in that it allows a class to read two different copies of the same logical data item. Our proof of serializability needs to be demonstrated at the logical level. We will first prove it on the level of physical data items, and then extend it to the logical level in Section 5.

v. c-readset:CLASSES -> 2^{DATA} defines the read-set of each class;

vi. c-writeset:CLASSES -> 2^{DATA} defines the write-set of each class.

An SDD-1 Execution History is a representation of the execution of a set of transactions for a particular database design. Formally, it is a seven-tuple <D, TRANS, classof, <, t-readset, t-writeset, LOG> where:

i. D = <DATA, DMs, stored-at, CLASSES, c-readset, c-writeset> is an SDD-1 database design;

ii. TRANS is a set of transactions, denoted {i,j,k,...};

iii. classof:TRANS -> CLASSES is a function denoting the class of each transaction (we will denote classof(i) as $\overline{i}$);

iv. < is a total order over TRANS. (In the SDD-1 implementation, i<j iff transaction i has a smaller timestamp than transaction j);

v. t-readset:TRANS -> DATA defines the read-set of each transaction such that for each i in TRANS, t-readset(i) is contained in c-readset(classof(i));

vi. t-writeset:TRANS -> DATA defines the write-set of each transaction such that for each i in TRANS, t-writeset(i) is contained in c-writeset(classof(i));

vii. LOG is an ordered pair <LOGELEMENTS,=> > such that

a. LOGELEMENTS = $LOGELEMENTS_R$ U $LOGELEMENTS_W$;

b. $LOGELEMENTS_R = \{R^i_{alpha}$: i is in TRANS, alpha is in DMs, and there is a data item x in t-readset(i) such that stored-at(x) = alpha};

c. $LOGELEMENTS_W = \{W^i_{alpha}$: i is in TRANS, alpha is in DMs, and there is a data item x in t-writeset(i) such that stored-at(x) = alpha};

d. => is a total order over LOGELEMENTS such that if there exists i in TRANS and alpha, beta in DMs with R^i_{alpha} and W^i_{beta} in LOGELEMENTS, then R^i_{alpha} => W^i_{beta}.

The definition of an SDD-1 execution history is based on several facts about SDD-1's operation:

1. The total order < on transactions follows from the fact that transactions are given globally unique timestamps (i.e., (iv) above).

2. The read-set and write-set of a transaction must be a subset of the read-set and the write-set of that transaction's class (i.e., (v) and (vi) above).

3. If data item x is in transaction i's read-set, and x is stored at alpha, then R^i_{alpha} must appear in LOG (i.e., (vii.b) above).

4. If data item x is in transaction i's write-set, and x is stored at alpha, then W^i_{alpha} must appear in LOG (i.e.,(vii.c) above).

5. For each transaction, all read operations must precede all write operations (i.e., (vii.d) above).

The total order < is used to determine how write operations affect the database. A write operation, say W^i_{alpha}, with x in t-writeset(i) actually writes a new value into x iff for all j with W^j_{alpha} => W^i_{alpha} and x in t-writeset(j), j < i. That is, W^i_{alpha} writes into x iff it is "later" than all previous transactions that wrote into x at alpha. This mechanism, which we call the Write Message Rule, is used to reorder write operations at each data module so that they appear to have occurred in "< -order" independent of their order of arrival [BERNSTEIN et al a,b].

2.2.2.3 Admissible Execution Histories

The concurrency control mechanism of SDD-1 consists of two components: a set of protocols, which are essentially procedures for processing a transaction's READ messages; and a set of protocol selection rules, which specify which protocols apply to transactions in each class (all transactions in a class use the same protocols). In a given execution history, if all transactions execute the protocols as specified by the protocol selection rules, then the execution history is called well-behaved. Intuitively, an execution history is well-behaved if all transactions follow the SDD-1 synchronization rules.

The main result of this report is that all well-behaved execution histories are serializable (see below). Since the motivation and formalism behind the SDD-1 concurrency control mechanism is based on some fundamental results about concurrency correctness in a database system, we must review these results before proceeding with a formal definition of well-behaved-ness and a proof of our theorem.

2.2.3 Serializable Histories

2.2.3.1 Consistency and Serializability

A prerequisite to proving the correctness of a system is a precise definition of what it means for the system to be correct. In SDD-1, we define the system to be correct if all possible histories are serializable. A serial history is one in which each transaction runs to completion before the next one starts. Thus, a serial history is one in which no concurrent activity has taken place. A serializable history is one which is equivalent to a serial history.

The intuitive justification for choosing serializability as the correctness criterion follows from the notion of consistency. Consistency may be considered to be a predicate over the state of the database; the database is either consistent or it is not. Each transaction submitted to the system is expected to preserve database consistency. That is, given a consistent database, it

will always produce a consistent database. A serial history, therefore, necessarily preserves consistency if all the transactions involved preserve consistency. Since a serializable history is equivalent to a serial one, then it too preserves consistency. The use of serializability as a correctness criterion is nearly universal [ESWARAN et al] [GRAY et al] and [HEWITT].

In order to define a serializable history as one which is equivalent to a serial history, we must be precise about what it means for two histories to be equivalent.

2.2.3.2 Equivalence

Intuitively, two histories are equivalent if they have the same effect on the database for all interpretations of transactions and all initial database states. (To account for I/O, we assume that all input and output operations are treated as distinct data items.) The notion of equivalence of histories is characterized by the reads-from relation [PAPADIMITRIOU et al].

Let H = <D, TRANS, classof, <, t-readset, t-writeset, LOG> be an execution history, where LOG=<LE, => >. Let R^i_{alpha} and W^i_{alpha} be elements of LE. We say that R^i_{alpha} reads x from W^j_{alpha} in H iff

1. $W^j_{alpha} \Rightarrow R^i_{alpha}$;
2. alpha = stored-at(x);
3. x is in t-readset(i);
4. x is in t-writeset(j); and
5. for all k in TRANS such that x is in t-writeset(k), if $W^k_{alpha} \Rightarrow R^i_{alpha}$ then $k < j$.

Intuitively, if R^i_{alpha} reads x from W^j_{alpha}, then the value of x read by R^i_{alpha} is the value of x produced by W^j_{alpha}. Part (5) of the definition ensures that no other W^k_{alpha} produced the x-value read by R^i_{alpha} (cf. the Write Message Rule at the end of Section 2.2). As a shorthand notation, if R^i_{alpha} reads x from W^j_{alpha}, then we also say that transaction i reads x from transaction j.

Not every read and write operation in H has an effect on the final database state produced by H. We call those transactions that do have an effect as live and define them as follows:

1. For each x in DATA, the "last" transaction in LOG that writes into x is live (i.e., if x is in t-writeset(k), and for all j with x in t-writeset(j) $W^j_{alpha} \Rightarrow W^k_{alpha}$ (where alpha = stored-at(x)), then k is live).

2. If transaction i is live, and for some x transaction i reads x from transaction j, then transaction j is live;

3. A transaction is live iff it so follows from (1) and (2).

A transaction that is not live is called dead.

Since every transaction at least prints something on an output device, no transaction is ever really dead. This can be modelled either by making each output operation write into a private write-set data item, or by simply assuming that all transactions are live. We make the latter assumption for the remainder of this report.

Two execution histories are equivalent if they have the same effect when applied to any database state. Formally, execution histories H_1 and H_2 are equivalent iff for every consistent database state S e domain(DATA) and for all interpretations of the transactions, H_1 and H_2 map S into the same final state, S_f. The following lemma characterizes equivalence of execution histories.

Lemma E [PAPADIMITRIOU et al 77]
Two histories, H_1 and H_2 are equivalent iff $TRANS_1$ and $TRANS_2$ have the same set of live transactions and for all

live i,j in $TRANS_1$, transaction i reads some data item x from transaction j in H_1 iff transaction i reads x from transaction j in H_2.

This is a standard program schema theoretic result, and can be found applied to a variety of models (e.g., [MANNA 74]). It can intuitively be justified by observing that if a transaction reads the same input data in both histories, then it will perform the same computation in both histories. The condition of Lemma E guarantees that each transaction reads the same inputs in both histories, thereby guaranteeing equivalence. The converse follows from the fact that the equivalence must hold over all interpretations of the transactions.

2.2.3.3 Serializability

Let A denote a symbol that is either an R or a W. Using this notation, we now define a log, LOG = <LE, => >, to be <u>serial</u> if there is no A^i_{alpha}, A^i_{beta}, A^j_{gamma}, A^j_{delta} in LE such that $i \neq j$, $A^i_{alpha} \Rightarrow A^j_{gamma}$ and $A^j_{delta} \Rightarrow A^i_{beta}$ (where A is a generic log symbol denoting an R or W). That is, a serial log is one in which no two transactions are interleaved. An execution history is <u>serial</u> iff its log is serial. As discussed earlier, serial execution histories are our benchmark for consistent executions.

An execution history is serializable if it is equivalent to some serial execution history. Since serial execution histories preserve database consistency, serializable execution histories preserve database consistency as well.

Given an execution history, H, we can determine whether or not H is serializable by defining a set of graphs on H called serialization graphs. To define serialization graphs on H, we introduce a live dummy transaction, called 'last', that executes after H and reads all data items (i.e., t-readset(last)=DATA). 'last' is defined to be live, and for all i in TRANS, i < last (where i≠last). The serialization graph on history H, denoted SG(L), is a *node*-labelled directed graph <V,E> where:

$V = \{i \mid \text{all } i \text{ in TRANS}\} + \{last\}$;

$E = E_{reads\text{-}from} + E_{interferes}$;

$E_{reads\text{-}from} = \{\langle i,j\rangle \mid j \text{ is live and } j \text{ reads some data item } x \text{ from } i\}$

$E_{interferes} = \{\langle k,i\rangle \mid$ for some j in TRANS, j is live, j reads some data item x from i, x is in t-writeset(k), and $k < i\}$ +
$\{\langle j,k\rangle \mid$ for some i in TRANS, j reads some data item x from i, x is in t-writeset(k), and $k > i\}$;

where '+' denotes set union. To simplify notation in the sequel, we will assume that all histories have a transaction 'last' at the end.

The edges in a serialization graph reflect the notion of "happened before." The edges in $E_{interferes}$ guarantee that if j reads x from i in H, then in the serialization of H no k that writes into x appears in a position that would have j read x from k (instead of from i).*

Theorem SER If SG(H) is acyclic then H is serializable.

Proof

Since SG(H) is acyclic, we can topologically sort its nodes, i.e. the elements of TRANS. This topological sort induces a serial log, say LOG', on the transactions in TRANS, i.e., LOG' is the sequence of serial transactions specified by the topological sort. Let H' be a history that differs from H only in that H' uses LOG' instead of LOG. If H' is equivalent to H, then H is serializable.

To show that H' is equivalent to H, we must show that for all i,j in TRANS, j reads some x from i in H iff j reads x

---

* 'last' is needed as a special case so that for each data item, x, the final transaction that successfully writes into x by the Write Message Rule is the same in LOG and in the serialization of LOG.

from i in H' (by Lemma E). Suppose j reads x from i in H. Then <i,j> is in $E_{\text{reads-from}}$ in SG(H) (denoted $E_{\text{reads-from}}$ (H)), so i precedes j in LOG'. By $E_{\text{interferes}}$ (H), for all k that also write into x, if k > i then j precedes k in LOG'. So, by the Write Message Rule, j reads x from i in H' as well. Conversely, suppose j reads x from i in H', but j reads x from k ($k \neq i$) in H. Then <k,j> is in $E_{\text{reads-from}}$ (H) and k precedes j in LOG'. If k < i, then <j,i> is in $E_{\text{interferes}}$ (H) and j precedes i in LOG'. But this means j cannot read x (or anything else) from i in H', a contradiction. If k > i, then <i,k> is in $E_{\text{interferes}}$ (H) and i precedes k in LOG'. By the Write Message Rule, j reads x from k in H', again a contradiction. So, j must read x from i in H also. Q.E.D.

An iff version of version of Theorem SER appears in [PAPADIMITRIOU et al] using a more general definition of serialization graph, but without the Write Message Rule. The latter forced us to reprove the theorem here.

Corollary SER If graph G contains all the edges of serialization graph, SG(H), and G is acyclic, then H is serializable.

2.2.4 Well-Behaved Execution Histories

2.2.4.1 Time-ordered Serialization Graphs

We define a relation over transactions in an execution history, H, called the time-ordered serialization graph, denoted by ->. For a given H, we define -> as follows:

1. $-> = ->_{rw} + ->_{wr} + ->_{ww}$

2. $i ->_{rw} j$ iff $i \neq j$ and there exists alpha in DMs such that $R^i_{alpha} => W^j_{alpha}$ and there exists some x for which stored-at(x) = alpha, such that x is in t-readset(i) and x is in t-writeset(j).

3. $i ->_{wr} j$ iff $i \neq j$ and there exists alpha in DMs such that $W^i_{alpha} => R^j_{alpha}$ and for some x with stored-at(x) = alpha, such that x is in t-writeset(i) and x is in t-readset(j).

4. i $->_{ww}$ j iff i<j and the intersection of t-writeset(i) and t-writeset(j) is non-empty.

The relation -> contains the serialization graph for H. That is, the graph $SG_0(H)$ defined as follows:

V_{SG_0} = {i| transaction i appears in H};

E_{SG_0} = {<i,j>| i -> j};

contains the serialization graph for H.

Lemma TOSG $SG_0(H)$ contains the serialization graph for H.

Proof

We must show that $E_{reads\text{-}from}$ and $E_{interferes}$ are included in the edge set of $SG_0(H)$.

To show that $E_{reads\text{-}from}$ is contained in $E_{SG_0(H)}$, we simply note that i-$>_{wr}$j subsumes $E_{reads\text{-}from}$.

To show that $E_{interferes}$ is contained in $E_{SG_0(H)}$, suppose R^j_{alpha} reads x from W^i_{alpha} in H (j ≠ last), and consider some other W^k_{alpha} that also writes into x. So, W^k_{alpha} and W^i_{alpha} intersect in x. If k < i, then k-$>_{ww}$i, so <k,i> is in $E_{SG_0(H)}$ as desired. So, suppose i < k. Since R^j_{alpha} reads from W^i_{alpha}, it follows that R^j_{alpha} => W^k_{alpha}. (If W^k_{alpha} => R^j_{alpha}, then R^j_{alpha} reads x from W^k_{alpha} rather than W^i_{alpha}, independent of the relative ordering of W^i_{alpha} and W^k_{alpha}.) Thus, j $->_{rw}$ k, so <j,k> is in

$E_{SG_0(H)}$ as desired. Hence, $E_{interferes}$ is contained in $E_{SG_0(H)}$. Q.E.D.

Corollary TOSG If $SG_0(H)$ is acyclic, then H is serializable.

Proof Follows directly from lemma TOSG and corollary SER. Q.E.D.

Execution histories, as defined in Section 2, may produce non-acyclic serialization graphs. However, if an execution history satisfies the SDD-1 synchronization rules, then cycles are not possible. In Section 4.2 we define precisely those execution histories that satisfy the SDD-1 synchronization rules. In Section 4.3 we show that all such histories produce acyclic serialization graphs. Combined with corollary TOSG, this is sufficient to show that such histories are serializable.

2.2.4.2 The SDD-1 Synchronization Mechanism

Intuitively, an execution history is "well-behaved" if each transaction in the history obeys the required protocols. The required protocols are determined by analyzing the database design, thereby assigning a set of protocols to each class. Each transaction is required to satisfy all of the protocols assigned to some class of which it is a new member. To formalize these concepts, we first explain what transactions must do to satisfy the protocols and then describe how protocols are assigned to classes of transactions.

There are four basic protocols in SDD-1: P1, P2, P3, and P4. A protocol is a property that an execution history must satisfy; it is implemented as an algorithm for executing read messages on behalf of transactions, which thereby only allows well-behaved histories to be produced.

Let H be an execution history. We say that H obeys protocol P1 with respect to classes $\overline{i}$ and $\overline{j}$ (written $P1(\overline{i},\overline{j})$) iff for each pair of transactions i and i' in $\overline{i}$ and j and j' in $\overline{j}$,

(P1) $j \rightarrow_{wr} i \leq i' \rightarrow_{rw} j'$ implies $j < j'$.

(Note: $i \leq i'$ means that either $i < i'$ or i is identical to i'.)

H obeys protocol P2 with respect to class $\bar{I}$ and a set of classes $J = \{\bar{j}_1, \ldots \bar{j}_n\}$ (written P2($\bar{I}$,J)) iff for each pair of transactions i and i' in $\bar{I}$ and j in $\bar{j}_u$ and j' in $\bar{j}_v$ (for some $\bar{j}_u$ and $\bar{j}_v$ in J)

(P2) $j \rightarrow_{wr} i \leq i' \rightarrow_{rw} j'$ implies $j < j'$.

H obeys protocol P3 with respect to classes $\bar{I}$ and $\bar{j}$ (written P3($\bar{I}$, $\bar{j}$)) iff for each transaction i in $\bar{I}$ and j in $\bar{j}$,

(P3) $i \rightarrow_{rw} j$ implies $i < j$

and $j \rightarrow_{wr} i$ implies $j < i$.

H obeys protocol P4 with respect to class $\bar{I}$ and a set of classes $J = \{\bar{j}_1, \ldots, \bar{j}_p\}$ (written P4($\bar{I}$, J)) iff for each transaction i in $\bar{I}$, j in $\bar{j}_u$, and j' in $\bar{j}_v$ (for some $\bar{j}_u$ and $\bar{j}_v$ in J),

(P4) $j \rightarrow j'$ and $i \leq j$ imply $i < j'$

and $j' \rightarrow j$ and $j \leq i$ imply $j' < i$.

The motivation for the protocols can be found in [BERNSTEIN et al b]. To review the uses of the protocols:

P1: implements pipelining;

P2: avoids having a read operation see updates in reverse timestamp order;

P3: avoids update race conditions;

P4: "cycle-breaking protocol," for unanticipated and very infrequent transactions.

Each transaction executes in a class. The protocols that each class must satisfy are determined from an analysis of the database design, using a mathematical structure called a conflict graph.

A conflict graph for a database design D = <DATA, DMs, stored-at, CLASSES, c-readset, c-writeset>, denoted CG(D), is an undirected node-labelled graph <V,E> where

$V = \{r^{\bar{i}} \mid \bar{i} \text{ in CLASSES}\} + \{w^{\bar{i}} \mid \bar{i} \text{ in CLASSES}\}$;

$E = E_{vert} + E_{horiz} + E_{diag}$;

$E_{vert} = \{\langle r^{\bar{i}}, w^{\bar{i}}\rangle \mid \bar{i} \text{ in CLASSES}\}$;

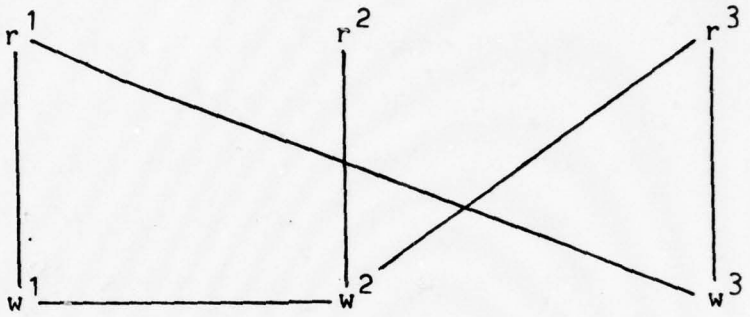
$E_{horiz} = \{\langle w^{\bar{i}}, w^{\bar{j}}\rangle \mid \bar{i},\bar{j}$ in CLASSES and the intersection of c-writeset($\bar{i}$) and c-writeset($\bar{j}$) is nonempty$\}$

$E_{diag} = \{\langle r^{\bar{i}}, w^{\bar{j}}\rangle \mid \bar{i},\bar{j}$ in CLASSES and the intersection of c-readset($\bar{i}$) and c-writeset($\bar{j}$) is nonempty$\}$

By convention, for each $\bar{i}$ the node $r^{\bar{i}}$ is drawn above $w^{\bar{i}}$ and for each $\bar{i}$ and $\bar{j}$ the nodes $r^{\bar{i}}$ and $r^{\bar{j}}$ are colinear on a horizontal line as are the nodes $w^{\bar{i}}$ and $w^{\bar{j}}$. This leads to the concepts of vertical, horizontal and diagonal edges (see Figure 2.1).

A Class Conflict Graph Figure 2.1

Class:	1	2	3
Readset:	{a}	{c}	{d}
Writeset:	{b}	{b,d}	{a}



A path in a conflict graph is a sequence of edges $[\langle i_0, i_1\rangle, \langle i_1, i_2\rangle, \langle i_2, i_3\rangle, \ldots, \langle i_{n-1}, i_n\rangle]$ taken from E. A cycle is a path in which $i_0 = i_n$. We call edges in E_{horiz} and E_{diag} heterogeneous, since they are incident with two distinct classes. A cycle is nonredundant if no more than two heterogeneous edges in the cycle are incident with any class (whether the class nodes are r's, w's or include one of each is not significant with respect to redundancy).

An execution history H on database design D is called well-behaved if it satisfies the following protocol selection rules:

I. For each nonredundant cycle, nrc, in CG(D) that contains a diagonal edge (i.e., $\langle r^{\overline{i}}, w^{\overline{j}}\rangle$), either

1. for some class $\bar{i}$ in the set of classes, J, that are incident with nrc, H obeys P4($\bar{i}$, J); or

2. each of the following hold:

   a. for all classes $\bar{i}$, $\bar{j}$, $\bar{k}$ such that $\bar{i} \neq \bar{j} \neq \bar{k}$, if edges $\langle r^{\bar{i}}, w^{\bar{j}} \rangle$ and $\langle r^{\bar{i}}, w^{\bar{k}} \rangle$ are in nrc, then H obeys P2($\bar{i}$,{$\bar{j}$, $\bar{k}$}); and

   b. for all classes $\bar{i}$, $\bar{j}$ such that $\langle r^{\bar{i}}, w^{\bar{j}} \rangle$ and either $\langle w^{\bar{j}}, w^{\bar{k}} \rangle$ or $\langle w^{\bar{j}}, r^{\bar{k}} \rangle$ (for some $\bar{k}$) are in nrc, then H obeys P3($\bar{i}$, $\bar{j}$).

II. For all classes $\bar{i}$, $\bar{j}$ such that $\langle r^{\bar{i}}, w^{\bar{j}} \rangle$ is in CG(D), H obeys P1($\bar{i}$, $\bar{j}$).

III. For all classes, $\bar{i}$, such that readset($\bar{i}$) intersects writeset($\bar{i}$), H obeys P3($\bar{i}$, $\bar{i}$).

2.2.4.3 Proof of Serializability

In corollary TOSG, we showed that if a particular serialization graph, $SG_0(H)$, defined on execution history H is acyclic, then H is serializable. We now complete the proof of serializability by showing that if H is well-behaved then $SG_0(H)$ is acyclic.

Lemma ACYC If execution history H is well-behaved, then $SG_0(H)$ is acyclic.

To prove this lemma, we need to show that the existence of a cycle in $SG_0(H)$ leads to a contradiction. As a preliminary step, we first examine special edge sequences in $SG_0(H)$, called trails. We will show that the endpoints of a trail, i and j say, are in timestamp order, i.e. $i < j$ (see lemma TRAIL below). Then, given any cycle in $SG_0(H)$, we show that for some transaction i in the cycle, there is a trail from i to i. But, by the previous result, this implies $i < i$, a contradiction. Hence, the cycle cannot exist. We proceed formally.

We define a trail in $SG_0(H)$ to be a sequence of n edges $\langle i_0, i_1\rangle, \langle i_2, i_3\rangle, \ldots, \langle i_{2n-2}, i_{2n-1}\rangle$ such that

1. n is greater than 0;

2. for j between 1 and n-1, $classof(i_{2j-1}) = classof(i_{2j})$ and $i_{2j-1} \leq i_{2j}$;

3. for j and k between 0 and n-1, with $j \neq k$, $classof(i_{2j}) \neq classof(i_{2k})$ and $classof(i_{2j+1}) \neq classof(i_{2k+1})$.

Lemma TRAIL Let H be a well-behaved execution history and let -> and $SG_0(H)$ be defined on H. Let $T = [\langle i_0, i_1\rangle, \langle i_2, i_3\rangle, \ldots, \langle i_{2n-2}, i_{2n-1}\rangle]$ be a trail in $SG_0(H)$ where $classof(i_0) = classof(i_{2n-1})$ and no transaction in the trail ran P4 with respect to the other classes in the trail. Then $i_0 < i_{2n-1}$.

Proof

Every edge in the trail corresponds to an edge in CG(D) in the following sense: for each j between 0 and n-1,

1. $i_{2j} \rightarrow_{wr} i_{2j+1}$ implies $\langle w^{\overline{i}_{2j}}, r^{\overline{i}_{2j+1}}\rangle$ is in CG(D);

2. $i_{2j} \rightarrow_{rw} i_{2j+1}$ implies $\langle r^{\overline{i}_{2j}}, w^{\overline{i}_{2j+1}} \rangle$ is in CG(D); and

3. $i_{2j} \rightarrow_{ww} i_{2j+1}$ implies $\langle w^{\overline{i}_{2j}}, w^{\overline{i}_{2j+1}} \rangle$ is in CG(D).

So, trail T corresponds to a sequence of edges, T_{CG}, in CG(D). We now prove two facts about T_{CG}.

Claim 1: The only cases in which T_{CG} contains two identical edges are when n = 2 and T_{CG} is of the form $[\langle r^{\overline{i}}, w^{\overline{j}} \rangle, \langle w^{\overline{j}}, r^{\overline{i}} \rangle]$, $[\langle w^{\overline{i}}, r^{\overline{j}} \rangle, \langle r^{\overline{j}}, w^{\overline{i}} \rangle]$, or $[\langle w^{\overline{i}}, w^{\overline{j}} \rangle, \langle w^{\overline{j}}, w^{\overline{i}} \rangle]$ where $i \neq j$.

Proof of claim 1: If n = 2, then the above forms are the only ones possible, such that T satisfies the definition of trail and T produces identical edges in T_{CG}. So, suppose T contains more than two edges, and two of its edges, say $\langle u, v \rangle$ and $\langle x,y \rangle$, produce identical edges in T_{CG}. By construction, all edges in T are "heterogeneous" (i.e., are incident with distinct classes), and therefore produce heterogeneous edges in T_{CG}. So, either the pairs u-x and v-y are each incident with the same class, or the pairs u-y and v-x are each incident with the same class. As long as T contains at least three edges, this implies that part(3) of the definition of trail is violated. (For example, if $\langle u, v \rangle$ and $\langle x, y \rangle$ are adjacent,

then the head of the edge preceding <u, v> is in the same class as y, and the tail of the edge following <x, y> is in the same class as u, thereby violating part (3) of the definition. Other cases follow by the same argument.)

Claim 2: For n greater than 2, T_{CG} can be augmented by homogeneous (i.e., vertical) edges to create a nonredundant cycle T'_{CG}.

Proof of claim 2: The head and tail of adjacent edges in T_{CG} are in the same class (by part(2) of the definition of trail). If they are not identical nodes, then they can be connected by a vertical edge (since they must be the r and w node of a single class). Insert all such vertical edges, creating a path T'_{CG}. No vertical edge in CG(D) will be added more than once (because of part (3) of the definition of trail). This fact, combined with claim 1, shows that T'_{CG} is a cycle. Part (3) of the definition of trail demonstrates that T'_{CG} is nonredundant.

Having set the stage with claim 2, we now proceed to prove the lemma by showing it to be true in each of the following three cases:

I. n = 1;

II. n = 2;

III. n greater than 2.

The cases subsume all possible trails T and therefore are sufficient to prove the lemma.

Case I Assume n = 1. Then T must be of the form [$<i_0, i_1>$]. Either $i_0 \rightarrow_{ww} i_1$ or $i_0 \rightarrow_{rw} i_1$ or $i_0 \rightarrow_{wr} i_1$. If $i_0 \rightarrow_{ww} i_1$, then $i_0 < i_1$ by definition of $\rightarrow_{ww}$. If $i_0 \rightarrow_{rw} i_1$ then t-readset(i_0) intersects t-writeset(i_1). So, the readset and writeset of $\bar{i}$ = classof(i_0) = classof(i_1) intersect. By the protocol selection rules, P3($\bar{i}$, $\bar{i}$) must be obeyed. Hence, $i_0 \rightarrow_{rw} i_1$ implies $i_0 < i_1$. If $i_0 \rightarrow_{wr} i_1$, then $i_0 < i_1$ follows by the same argument.

Case II: Assume n = 2. Then T must be of the form [$<i_0, j_0>$, $<j_1, i_1>$]. Let $\bar{i}$ = classof(i_0) = classof(i_1) and $\bar{j}$ = classof(j_0) = classof(j_1). There are nine subcases to consider, depending on the way the i's are related by ->. Note that $j_0 < j_1$ by definition of trail.

In the first three subcases, we assume $i_0 \rightarrow_{ww} j_0$. By definition of $\rightarrow_{ww}$, we have $i_0 < j_0$. Since $j_0 < j_1$, by transitivity $i_0 < j_1$.

Subcase a: Suppose $j_1 \rightarrow_{ww} i_1$. By definition of $\rightarrow_{ww}$, $j_1 < i_1$. So, by transitivity $i_0 < i_1$.

Subcase b: Suppose $j_1 \rightarrow_{rw} i_1$. Then T'_{CG} is the nonredundant cycle $[\langle w^{\bar{i}}, w^{\bar{j}}\rangle, \langle w^{\bar{j}}, r^{\bar{j}}\rangle, \langle r^{\bar{j}}, w^{\bar{i}}\rangle]$. By the protocol selection rules, $P3(\bar{j}, \bar{i})$ must be obeyed. Hence, $j_1 \rightarrow_{rw} i_1$ implies $j_1 < i_1$, and by transitivity $i_0 < j_1$.

Subcase c: Suppose $j_1 \rightarrow_{wr} i_1$. Then T'_{CG} is the nonredundant cycle $[\langle w^{\bar{i}}, w^{\bar{j}}\rangle, \langle w^{\bar{j}}, r^{\bar{i}}\rangle, \langle r^{\bar{i}}, w^{\bar{i}}\rangle]$. By the protocol selection rules, $P3(\bar{i}, \bar{j})$ must be obeyed. Hence, $j_1 \rightarrow_{wr} i_1$ implies $j_1 < i_1$, and by transitivity $i_0 < i_1$.

In the next three subcases (d,e,f), assume $i_0 \rightarrow_{wr} j_0$.

Subcase d: Suppose $j_1 \rightarrow_{ww} i_1$. Then $j_1 < i_1$ and T'_{CG} is the nonredundant cycle $[\langle w^{\bar{i}}, r^{\bar{j}}\rangle, \langle r^{\bar{j}}, w^{\bar{j}}\rangle, \langle w^{\bar{j}}, w^{\bar{i}}\rangle]$. By the protocol selection rules, $P3(\bar{j}, \bar{i})$ must be obeyed. Hence, $i_0 \rightarrow_{wr} j_0$ implies $i_0 < j_0$, and by transitivity $i_0 < i_1$.

Subcase e: Suppose $j_1 \rightarrow_{rw} i_1$. By the protocol selection rules, $P1(\bar{j}, \bar{i})$ must be obeyed. Hence, $i_0 \rightarrow_{wr} j_1 < j_1 \rightarrow_{rw} i_1$ implies $i_0 < i_1$.

Subcase f: Suppose $j_1 \rightarrow_{wr} i_1$. Then T'_{CG} is the nonredundant cycle $[\langle w^{\bar{i}}, r^{\bar{j}}\rangle, \langle r^{\bar{j}}, w^{\bar{j}}\rangle, \langle w^{\bar{j}}, r^{\bar{i}}\rangle, \langle r^{\bar{i}}, w^{\bar{i}}\rangle]$, and both $P3(\bar{i}, \bar{j})$ and $P3(\bar{j}, \bar{i})$ must be obeyed. $P3(\bar{j}, \bar{i})$ and $i_0 \rightarrow_{wr} j_0$ implies $i_0 < j_0$. $P3(\bar{i}, \bar{j})$ and $j_1 \rightarrow_{wr} i_1$ implies $j_1 < i_1$. So, by transitivity, $i_0 < i_1$.

In the final three subcases (g,h,i), assume $i_0 \rightarrow_{rw} j_0$.

Subcase g: Suppose $j_1 \rightarrow_{ww} i_1$. Then $j_1 < i_1$ and T'_{CG} is the nonredundant cycle $[\langle r^{\bar{i}}, w^{\bar{j}}\rangle, \langle w^{\bar{j}}, w^{\bar{i}}\rangle, \langle w^{\bar{i}}, r^{\bar{i}}\rangle]$. So, P3($\bar{i}$, $\bar{j}$) must be obeyed. Since $i_0 \rightarrow_{rw} j_0$, $i_0 < j_0$. Hence, by transitivity, $i_0 < i_1$.

Subcase h: Suppose $j_1 \rightarrow_{rw} i_1$. This case is essentially the same as subcase f.

Subcase i: Suppose $j_1 \rightarrow_{wr} i_1$. Then, P1($\bar{j}$, $\bar{i}$) must be obeyed. We assume $i_1 < i_0$ and show a contradiction. This follows directly, since $j_1 \rightarrow_{wr} i_1 < i_0 \rightarrow_{rw} j_0$ implies $j_1 < j_0$, contradicting $j_0 < j_1$.

Case III Assume that n is greater than two. We define a class, $\bar{i}$, to be a P2-class in T if there are transactions i and i' in $\bar{i}$ that appear in T in edges of the form $j \rightarrow_{wr} i$ and $i' \rightarrow_{rw} k$, for some transactions j and k. (Note that classof(i_0) = classof(i_{2n-1}) can be a P2-class.) We first prove two preliminary claims about T.

Claim 3: Let $[\langle j_0, j_1\rangle, \ldots, \langle j_{2m-2}, j_{2m-1}\rangle]$ be a sequence of edges in T such that no j_k is in a P2-class of T. Then $j_0 < j_{2m-1}$.

Proof of claim 3: We prove the claim by induction on the number of edges in the sequence. As the basis, we show j_0

$< j_1$. Then we show that after adding an edge to the prefix $[\langle j_0, j_1\rangle, \ldots, \langle j_{2p-2}, j_{2p-1}\rangle]$, where p is smaller than m, if $j_0 < j_{2p-1}$, then $j_0 < j_{2p+1}$.

Basis step: Either $j_0 \rightarrow_{ww} j_i$, $j_0 \rightarrow_{wr} j_1$, or $j_0 \rightarrow_{rw} j_1$. If $j_0 \rightarrow_{ww} j_1$, then $j_0 < j_1$ follows directly from the definition $\rightarrow_{ww}$. Suppose $j_0 \rightarrow_{wr} j_1$. Then the diagonal edge $\langle w^{\overline{j}}0, r^{\overline{j}}1\rangle$ appears in T'_{CG} and, with claim 2, T'_{CG} is a nonredundant cycle with a diagonal edge. Since j_1 is not in a P2-class, the edge following $\langle j_0, j_1\rangle$ (or edge $\langle i_0, i_1\rangle$, if $j_1 = i_{2n-1}$) must be an $\rightarrow_{ww}$ or $\rightarrow_{wr}$. So, P3($\overline{j}_1$, $\overline{j}_0$) must be obeyed. Hence, $j_0 \rightarrow_{wr} j_1$ implies $j_0 < j_1$. Suppose $j_0 \rightarrow_{rw} j_1$. The diagonal edge $\langle r^{\overline{j}}0, w^{\overline{j}}1\rangle$ appears in T'_{CG} and, with claim 2, T'_{CG} is a nonredundant cycle with a diagonal edge. Since $\overline{j}_0$ is not a P2-class, the edge preceding $\langle j_0, j_1\rangle$ (or edge $\langle i_{2n-2}, i_{2n-1}\rangle$ if $j_0 = i_0$) must be an $\rightarrow_{ww}$ or $\rightarrow_{rw}$. So, P3($\overline{j}_0$, $\overline{j}_1$) must be obeyed. Hence, $j_0 \rightarrow_{rw} j_1$ implies $j_0 < j_1$.

Induction step: Suppose $[\langle j_0, j_1\rangle, \ldots, \langle j_{2p-2}, j_{2p-1}\rangle]$ has $j_0 < j_{2p-1}$. If p = m, then the claim is proved. Else, augment the sequence by the edge $\langle j_{2p}, j_{2p+1}\rangle$. By the same argument as the basis step, $j_{2p} < j_{2p+1}$. By definition of trail, $j_{2p-1} < j_{2p}$. So, by transitivity, $j_0 < j_{2p+1}$, thereby proving the claim.

Claim 4: Let $[\langle j_0, j_1\rangle, \langle j_2, j_3\rangle]$ be a sequence of edges in T such that $j_0 \rightarrow_{wr} j_1$ and $j_2 \rightarrow_{rw} j_3$. Then $j_0 < j_3$.

Proof of claim 4: Since T'_{CG} contains the diagonal edge $\langle r^{\bar{j}}0, w^{\bar{j}}1\rangle$, the protocol selection rules imply that $P2(\bar{j}_1, \{\bar{j}_0, \bar{j}_3\})$ must be obeyed. That $j_0 < j_3$ follows by definition of P2.

We can prove case III for two subcases: $\bar{i}_0$ is a P2-class and $\bar{i}_0$ is not a P2-class.

Suppose $\bar{i}_0$ is a P2-class. Consider T with its first and last edges removed (call it T"), T" = $[\langle i_2, i_3\rangle, \ldots, \langle i_{2n-4}, i_{2n-3}\rangle]$. (Since n is greater than two, there must be at least one edge in between.) T" consists of sequences with no P2-class transactions (as per claim 3) separated by sequences of P2-class transaction pairs. By claims 3 and 4, the left and right endpoints of each of these sequences satisfy the relation left-endpoint < right-endpoint. So, by repeated application of transitivity, we have $i_2 < i_{2n-3}$. Since $\bar{i}_0$ is a P2-class, $P2(\bar{i}_0, \{\bar{i}_2, \bar{i}_{2n-3}\})$ must be obeyed. By examining T, we have $i_0 \rightarrow_{rw} i_1 \leq i_2 < i_{2n-3} \leq i_{2n-2} \rightarrow_{wr} i_{2n-1}$. Suppose $i_{2n-1} < i_0$. Then $i_{2n-2} \rightarrow_{wr} i_{2n-1} < i_0 \rightarrow_{rw} i_1$ implies (by P2) that $i_{2n-2} < i_1$. But $i_1 \leq i_2 < i_{2n-3} \leq i_{2n-1}$ implies $i_1 < i_{2n-1}$, a contradiction. So $i_0 < i_{2n-1}$.

Suppose $\bar{i}_0$ is not a P2-class. Then T has exactly the same form as T" above. So, by repeated application of claims 3 and 4 and transitivity, we obtain $i_0 < i_{2n-1}$ as desired. This proves case III and the lemma. Q.E.D.

Lemma PATH Let H be a well-behaved execution history and let -> and $SG_0(H)$ be defined on H. Let P = $[\langle i_0, i_1\rangle,\ldots,\langle i_{2n-2}, i_{2n-1}\rangle]$ be a path in $SG_0(H)$, where classof(i_0) = classof(i_{2n-1}). Then $i_0 < i_{2n-1}$.

Proof

First, suppose some transaction, i_{2p}, in P ran protocol P4 with respect to the other classes in the path. That is, P = $[\langle i_0, i_1\rangle,\ldots,\langle i_{2p}, i_{2p+1}\rangle,\ldots,\langle i_{2n-2}, i_{2n-1}\rangle]$. By definition of P4, i_{2p} -> i_{2p+1} implies $i_{2p} < i_{2p+1}$. Similarly, for each q greater than p, $i_{2p} < i_{2q-1}$ and $i_{2q-1} = i_{2q}$ and i_{2q} -> i_{2q+1} implies $i_{2p} < i_{2q+1}$. So, by induction, $i_{2p} < i_{2n-1}$. A similar argument shows $i_0 < i_{2p}$. (Note: possibly $i_0 = i_{2p}$ or $i_{2n-1} = i_{2p}$, but not both.) Hence, by transitivity, $i_0 < i_{2n-1}$.

So, assume no transaction in path P ran protocol P4 with respect to the other classes in the path. Our proof is by induction on n. As the basis step, assume that n = 1. Then P is a trail, and $i_0 < i_{2n-1}$ follows from lemma TRAIL.

Assume the lemma is true for all n less than k. We now show it to be true for n = k. Suppose, for some q between 1 and n-1, classof(i_{2q}) = classof(i_0). Then P can be partitioned into two paths [$\langle i_0, i_1\rangle,\ldots,\langle i_{2q-2}, i_{2q-1}\rangle$], and [$\langle i_{2q}, i_{2q+1}\rangle,\ldots,\langle i_{2-2}, i_{2n-1}\rangle$]. By the inductive assumption, $i_0 < i_{2q-1}$ and $i_{2q} < i_{2n-1}$, since the subpaths are of length less than k. So, $i_0 < i_{2n-1}$ by transitivity.

Suppose no such q exists. Then choose some class $\overline{j}$ that contains transactions j' and j" in P that are connected by a nonempty path. That is, P is of the form [$\langle i_0, i_1\rangle,\ldots,\langle j', \rangle,\ldots,\langle ,j"\rangle,\ldots,\langle i_{2n-2}, i_{2n-1}\rangle$]. Excise the path from j' to j" from P. By the induction assumption, j' < j". Excise all such paths that connect two transactions in the same class. When no such transaction pairs remain (except those that are adjacent), the resulting sequence of edges is a trail. (Adjacent edges are incident with two transactions from the same class that are either identical or are related by increasing <. Also, no class appears in more than two heterogeneous edges.) Now, by lemma TRAIL, $i_0 < i_{2n-1}$, as desired. Q.E.D.

Lemma ACYC follows as a corollary to lemma PATH.

Lemma ACYC If execution history H is well-behaved, then $SG_0(H)$ is acyclic.

Proof

Suppose there is a path in $SG_0(H)$ from transaction i back to itself. By lemma PATH, $i < i$. But $<$ is a total order, a contradiction. Q.E.D.

We may now state and prove the main theorem of this section:

Theorem SR: If all execution histories produced by SDD-1 are well-behaved, then they are all serializable.

Proof: Follows directly from corollary TOSG and lemma ACYC. Q.E.D.

2.2.5 Serializability of Logical Transactions

The formalism which has been presented so far deals with transactions which have read-sets and write-sets of physical data items. In practice, however, the user of SDD-1 expresses his transactions in terms of logical data items. A logical data item may correspond to a number of physical copies stored in the database, presumably all at different sites. SDD-1 maps user transactions expressed in terms of logical data items into transactions referring to physical data items according to the following rule. When a logical data item is read the system chooses one of the physical copies to read. However, when a logical data item is written, the system updates every physical copy of the logical data item.

We would like to prove that SDD-1 generates a serializable history of logical transactions against a logical database. That is, the transactions appear to be serializable against a hypothetical database in which there is only one copy of each logical data item. Furthermore, we wish to show that the Write Message Rule is not needed in the serialization, i.e. all write

messages always apply all of their updates in the serialization. That is, updates specified in the user's transaction always update the database, and cannot be accidently ignored due to the Write Message Rule.

We could extend our formalism to include the notions of logical transaction, logical data item, a logical to physical data item mapping, and a correctness definition based on logical transactions rather than physical transactions. Instead of actually developing this additional mechanism however we will simply present an informal plausibility argument for the serializability of logical transactions.

The argument is a simple one and goes as follows. Consider the serialized execution history corresponding to an actual interleaved history. In between completely executed transactions in this serial history, all physical copies of each logical data item have the same value. This follows because any transaction which updates one copy must update all the others as well. Thus, since all physical copies have the same value, the behavior of the system is the same as if there were only one physical copy corresponding to each logical data item. And further, since the actual interleaved history is defined to be equivalent to the serialized history, it too behaves as if each logical data item had only a single physical copy.

Finally, we note that whenever a write-write intersection occurs between transactions, the system serializes those transactions in <- order. Thus the serial history behaves as if all updates applied unconditionally to the database, without reference to the Write Message Rule. And thus, by equivalence, all transaction updates actually affect the database.

2.3 Initial SDD-1 Implementation

During this reporting period, the first working version of SDD-1 was implemented and demonstrated. This version of the system implements the distributed query processing algorithm developed by CCA and described in [WONG] and [CCA b]. In addition, the system was enhanced to include a graphics interface which visually illustrates the distributed processing and data movement as it occurs during the execution of a distributed query. The remainder of this section describes the initial system architecture, the query processing algorithm and the demonstration setup.

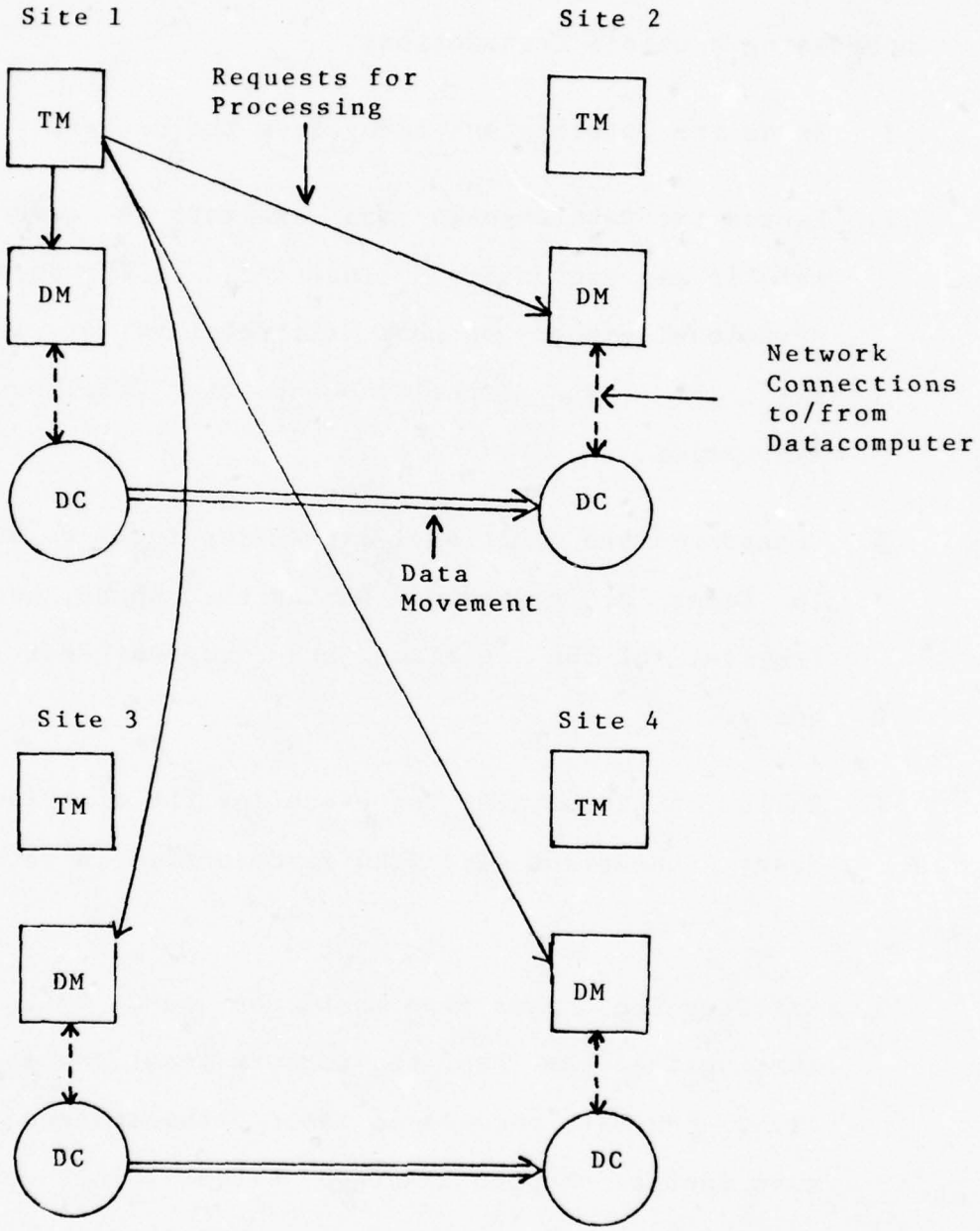
2.3.1 System Architecture

The current configuration of SDD-1 involves four computers (sites) on the ARPANET. Two of the machines run TENEX and two run TOPS-20 operating systems. One to three datamodules (DMs) and one transaction module (TM) run at each site. Queries to the system are entered at a TM. The TM is responsible for determining the access strategy to efficiently execute the user's query, based on the distribution of data within the system. The TM then requests one or more DMs to perform local processing and/or move sub-relations from one site to another. Each DM is connected to a Datacomputer subjob which performs local database management functions. The number of DMs per site is parameterized. As the number of DMs per site increases, the amount of potential parallelism increases.

The entire system runs under MSG ([BBN]), the interprocess communications system developed by BBN for the NSW. As far as SDD-1 is concerned, MSG simply provides a pipeline from the TM to each DM with which it interacts. The DMs communicate with their Datacomputer subjobs through Arpanet connections. In addition, during the execution of

a query, Datacomputers at different sites establish Arpanet connections for the purpose of moving data from site to site. Figure 2.2 illustrates the system configuration with one DM per site. The figure includes some of the relevant communications paths in the system.

SDD-1 Implementation Modules
Figure 2.2



The most interesting module in the system, insofar as SDD-1 is concerned, is the TM. The TM is at the heart of the system. It performs the following operations while processing a user's transaction:

1. Reads the Datalanguage query from the network.

2. Parses the Datalanguage and extracts a covering relational expression. That is, it produces a relational expression that will retrieve at least all the data specified by the Datalanguage expression.

3. Transforms the relational expression into a query in terms of fragments. During this phase, unused fragments of the relation are dropped from the query.

4. Builds an access plan for executing the distributed query (the exact algorithm is described in section 2.3.2).

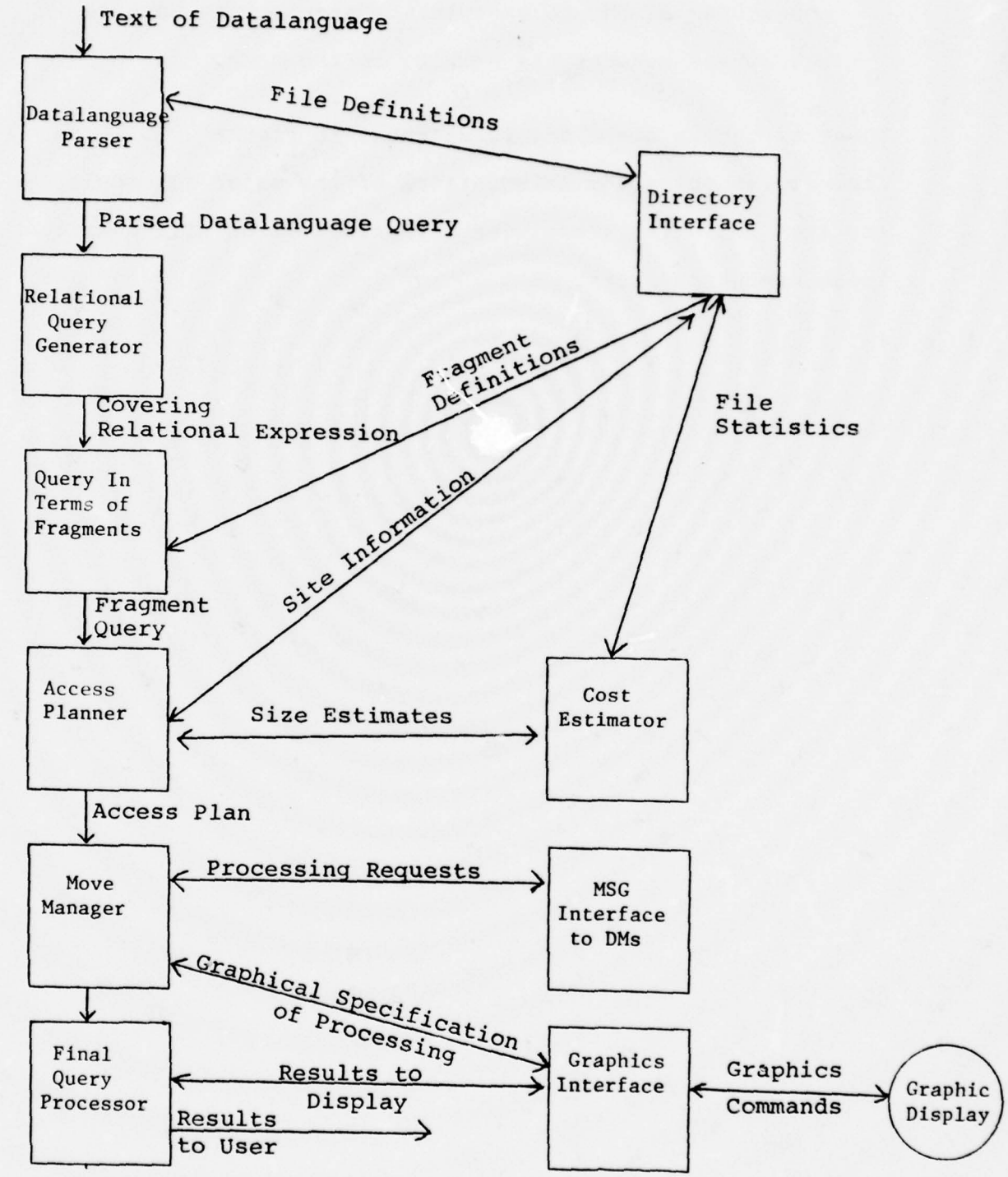
5. Executes the access plan that was just built by instructing the DMs to perform local processing (i.e. run Datalanguage on their Datacomputers) and move partial results to other sites. This phase may also cause a graphical representation of the TM's instructions to a DM and the movement of data between DMs to be drawn on the display.

6. Finally, when all the required data is at one site, requests a DM to execute the user's Datalanguage query and returns the results to the user.

These execution steps are illustrated in figure 2.3. The figure also shows the interactions of the major submodules in the TM. The remainder of this section will detail these execution steps.

Major Submodules of the TM

Figure 2.3



2.3.1.1 User Interface

A user program communicates with SDD-1 in the same manner as it communicates with the Datacomputer. Datalanguage requests are sent to the TM through a network connection, and messages and results are returned to the user through another network connection. SDD-1 was purposely designed to appear to the user program identical to the Datacomputer so that existing Datacomputer applications programs would not have to be modified for use in the SDD-1 environment.

2.3.1.2 Relational Query Generation

Query optimization is most convenient if the query is expressed in a non-procedural data access language. Since the query language of SDD-1, Datalanguage, is semi-procedural, the TM translates the Datalanguage into a relational query. The Datalanguage parser and relational query generator modules take the text of the user's Datalanguage query and produce a relational query that is guaranteed to retrieve at least all the data required by the user query.

In the process of generating the relational query, the directory interface is accessed to verify that the appropriate files and fields exist and that the user is granted access to the data. If any of these conditions are not met, an error message is sent back to the user and the query is aborted. In the current version of the system, the directory system simply accesses a preloaded cache. The distributed directory scheme described in section 2.1.5, [ROTHNIE and GOODMAN], and [CCA a] will be implemented in future versions of the system.

2.3.1.3 Fragment Transformation

The unit of data distribution in SDD-1 is the fragment of a relation. If a relation is thought of as a rectangular table, a fragment can be considered to be a rectangular subset of that table. In SDD-1 fragments are formed by first splitting the relation horizontally into sets of tuples based on predicates and then splitting the resulting fragments vertically by projecting them on non-overlapping subsets of the fields. In addition, a user invisible key called the tuple identifier (TID) is replicated in each vertical fragment to guarantee that each sub-tuple is uniquely linked with the other sub-tuples that comprise the logical tuple.

The fragment transformation section of the system is responsible for transforming the relational query into one that deals with the stored fragments as opposed to the logical relations. In the process of doing this transformation, the first step of access planning is taken by eliminating fragments that are not required by this query.

The transformation algorithm works in the following way. For each relation referenced in the query, a horizontal fragment transformation is performed followed by a vertical fragment transformation. The horizontal transformation consists of splitting the query into a series of sub-queries, one for each horizontal fragment and specifying that the results of these queries will be appended to produce the final result. During the transformation process, the query's restriction is tested against the fragment definition predicate to see if the particular fragment is included or excluded by the restriction. For example, if a fragment is defined on a SHIP file as those tuples with NAT field equal to 'US', and if the users query requests tuple with NAT equals 'USSR', then the fragment in question may be completely dropped from the query.

Each horizontal fragment query is transformed into a series of sub-queries in terms of its constituent vertical fragments. This is accomplished by causing each field in the query's predicate and target list to refer to the particular vertical fragment in which it resides. Any vertical fragments that are not referenced are dropped from the query. In addition, joining terms based on TID are added to the sub-queries to assure that a set of fields from more than one vertical fragment is treated as the same logical tuple.

2.3.1.4 Access Planner

The access planner takes the transformed query and, using the algorithm described in section 2.3.2, produces a plan for executing the query. In order to formulate its plan, the access planner interfaces with the directory system and the cost estimation module. The directory provides statistical information for each field concerning the distribution of values as well as the location (site) of each stored fragment. The cost estimation module, using the statistical information provided by the directory, estimates the sizes of results based on the query's restrictions and estimates local processing costs.

The result of the access planner is a table specifying the steps required to retrieve the necessary data. The table contains one entry for each site involved in the query. Each entry contains one or more specifications of local processing that will occur at that site. In addition, each local processing specification includes the destination site for its results.

2.3.1.5 Move Manager

The task of the move manager is to direct the execution of the plan built by the access planner. In addition, if the demonstration display is in use, the move manager is responsible for communicating local processing steps and data movements to the graphics interface.

The move manager's algorithm is very straight-forward. For each site, it finds all the local processing that is "runnable". Local processing is ready to run when all required data is present at its site and both the sending and receiving sites each have a DM available for processing. Since each site has a limited number of DMs (usually three), it is conceivable that some of the local processing might not be runable even though all required data is present.

The move manager initiates local processing and movement of data between two Datacomputers by sending the appropriate messages to the corresponding DMs. The sender's Datalanguage is constructed by transforming the relational expression associated with the local processing into Datalanguage. As local-processing/data-movement steps finish, new files which contain the moved sub-relations are created at the destination sites. When the entire plan has been executed, the move manager has finished its job.

The move manager modifies its behavior slightly if the demonstration display is being used. As local processing is initiated, the relational expression, the source site and the destination site are sent to the graphics interface. The graphics interface is notified when a particular pair of sites have completed their data movement. These calls to the graphics interface cause graphical representations of the processing and data movement to appear on the screen.

2.3.1.6 Final Query Processing

Once all relevant data has been moved to one site, the system is ready to perform the processing requested by the user. The original query submitted by the user is sent to the final site. To complete the processing, the results of the final query are shipped through the TM to the user. In demonstration mode, the results are also displayed on the graphics terminal.

2.3.2 Distributed Query Processing

The goal of a distributed query processing algorithm is to get all the requested data to the user's site. There are a number of ways to accomplish this goal. One approach would be to adapt a single-site query processing algorithm by replacing disk accesses with accesses to foreign sites. Whenever the query processor wanted some data from the disk, the system would trap the access and divert it to the network to retrieve data from a remote site. A second approach might be to change the tuple accessing code of a centralized query processing system to retrieve non-local tuples from the network as they are needed.

Both of the above approaches to distributed query processing are flawed because they fail to take into account the relative low speed of communications channels in a computer network. These channels are typically several orders of magnitude slower than disk channels. The first of the above two approaches would be unacceptably slow because of the amount of data transferred. The second approach seems more intelligent in that it accesses only the required data. However, it would be slow because it would be swamped by the number of messages being sent over the network. In a computer network there is a fixed overhead associated with each message in addition to the time required to transmit the actual data.

The shortcomings of the above approaches suggest that a good distributed query processing algorithm should try to minimize both the amount of data moved from site to site and also minimize the number of messages sent between sites. In addition, the algorithm should take advantage of the computing power at all of the sites involved while processing the query.

The SDD-1 distributed query processing algorithm uses two tactics: local processing and reducing moves. Local processing consists of performing restrictions and

projections on data at a single site with the objective of reducing the amount of data moved. This tactic is almost always profitable since communications costs almost always outweigh local processing costs. The tactic of reducing moves is the movement of small amounts data between the local data management systems, if it helps reduce the amount of data that must be moved later. Even though reducing moves increase the total number of messages needed to execute the query, they are still desirable if they yield a significant enough reduction in the total time to obtain the desired result. The optimization problem consists of finding the best sequence of local processing and reducing moves. The remainder of this section describes the optimization algorithm.

The strategy for executing a query is produced by selecting an initial strategy and perterbing it to gain improvements. The initial strategy is to perform all the local processing possible (in parallel) at each site, in order to restrict the referenced fragments. The site with the largest amount of data after local processing is chosen as the final site. The reduced data is moved from all other sites to the final site. This minimizes the amount of data moved between sites based solely on initial local processing. The initial strategy may be optimized if two fragments (one small and one large), residing at

different sites, are linked by a joining term and both must be moved to the final site. The optimization involves moving the smaller of the two fragments to the site of the larger one; followed by a movement of both of the fragments, further restricted by the joining term, to the final site. The initial move is replaced by the reducing move, followed by the movement of the join, only if the total time required for the two moves (of potentially much less data) is an improvement over the initial move. This algorithm is applied to the new set of moves until there are no profitable reducing moves to be made.

In order to compute the cost of a move, the system must be able to estimate the size of a relation if it is restricted, projected or joined with another relation. This is accomplished by maintaining statistics in the directory for each horizontal fragment: the number of tuples in the fragment, the number of unique values found in each non-numeric field and the minimum and maximum values for numeric fields. This is enough information to make simple estimates of result sizes if one assumes evenly distributed field values and independence of fields.

The query processing algorithm as presented suffers from the usual hill-climbing problem in that it may find a local optimum. While this problem is inherent in the approach, it is alleviated somewhat by a branch-and-bound technique that considers many possible final sites. This technique rejects a possible final site if its best case cost is greater than some other site's worst case cost. Experience has shown that this technique produces better results in many situations.

2.3.3 The Demonstration Setup

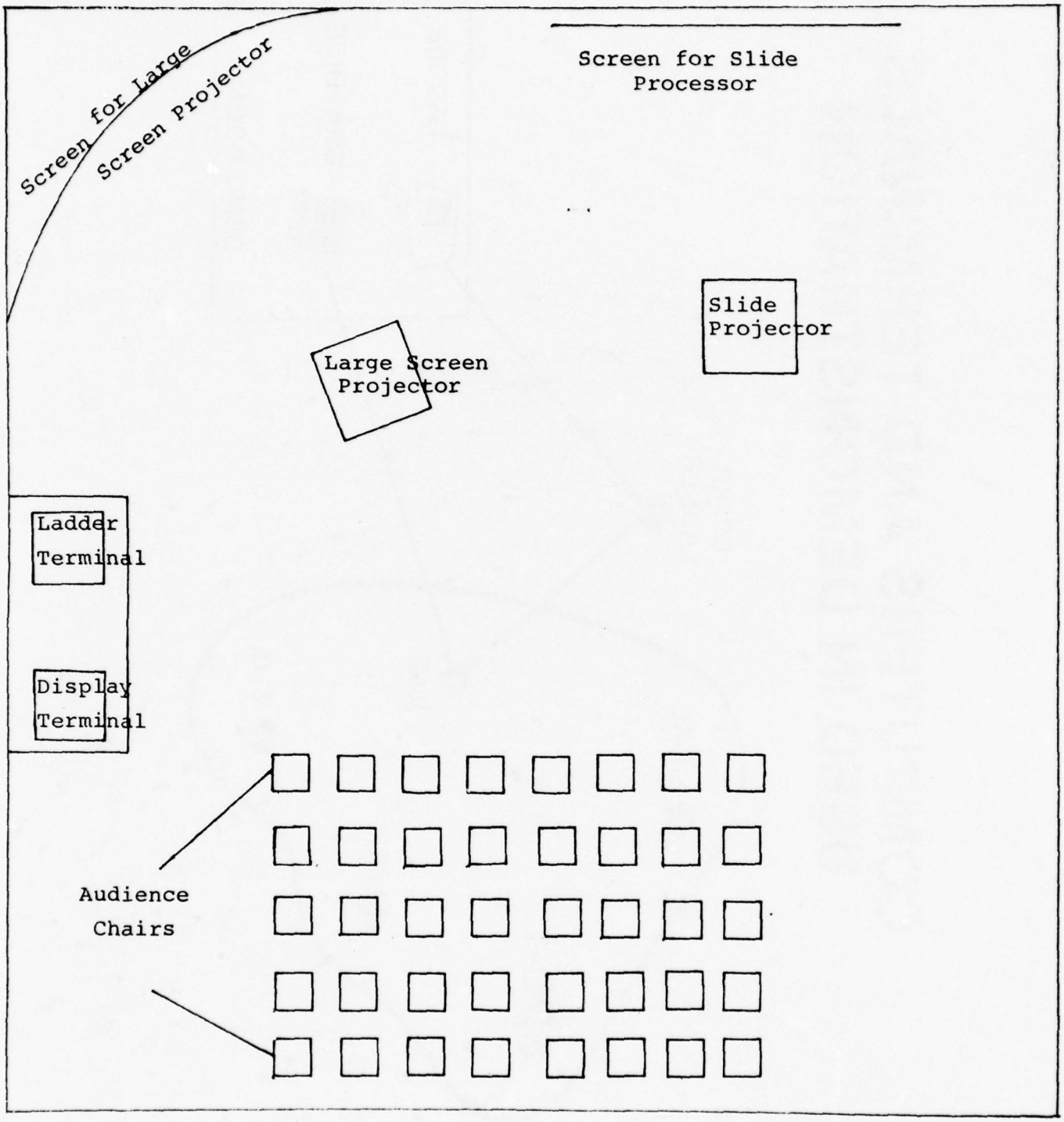
The initial implementation of SDD-1 includes a unique graphical representation of the query processing algorithm in action. The major problem in demonstrating a distributed system like SDD-1 is that the system appears to the user as if it were a centralized DBMS. The user is completely hidden from the distributed aspects of the system. Whereas in a real working environment hiding the distribution aspects of the system is desirable, in a demonstration environment the opposite is often true in that it is the distributed algorithm that is being demonstrated. The remainder of this section describes the physical demonstration environment, the SDD-1

configuration during a demo and an example of the output produced by the demo.

An SDD-1 demonstration is a combination of a briefing with slides and a live presentation of the system in operation. In a typical demonstration setup there is an overhead projector and screen for the briefing and a color graphics terminal (a Ramtek Micrographics Terminal) for the live demo. In addition, depending on the size of the audience, there may be a large screen projector (such as an Advent 1000B) connected to the display terminal. The entire setup is completed by an additional terminal where queries are entered to the system. Figure 2.4 shows a typical setup for a group of 20 to 30 people.

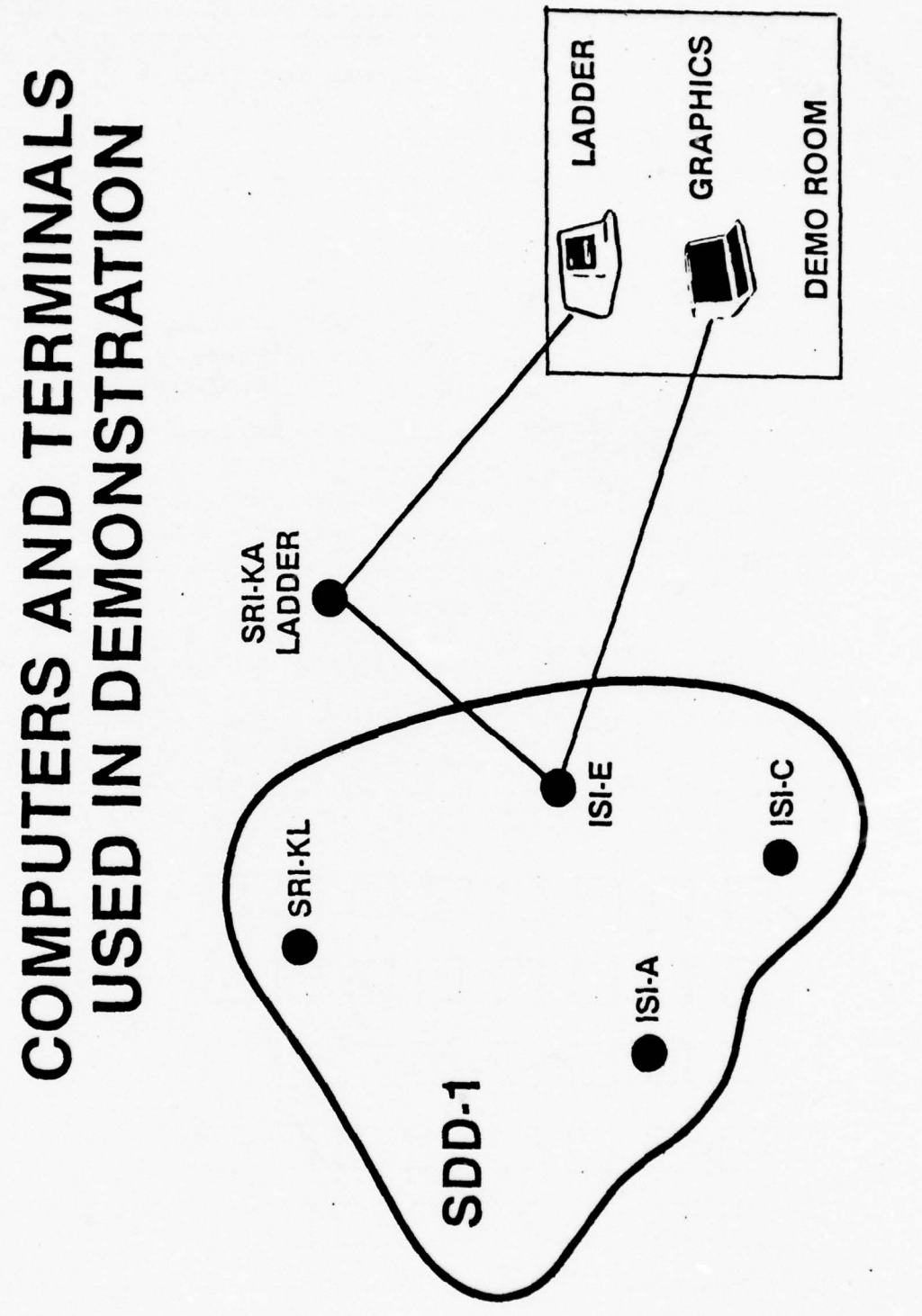
Demonstration Room Setup

Figure 2.4



SDD-1 Computers and Terminals

Figure 2.5



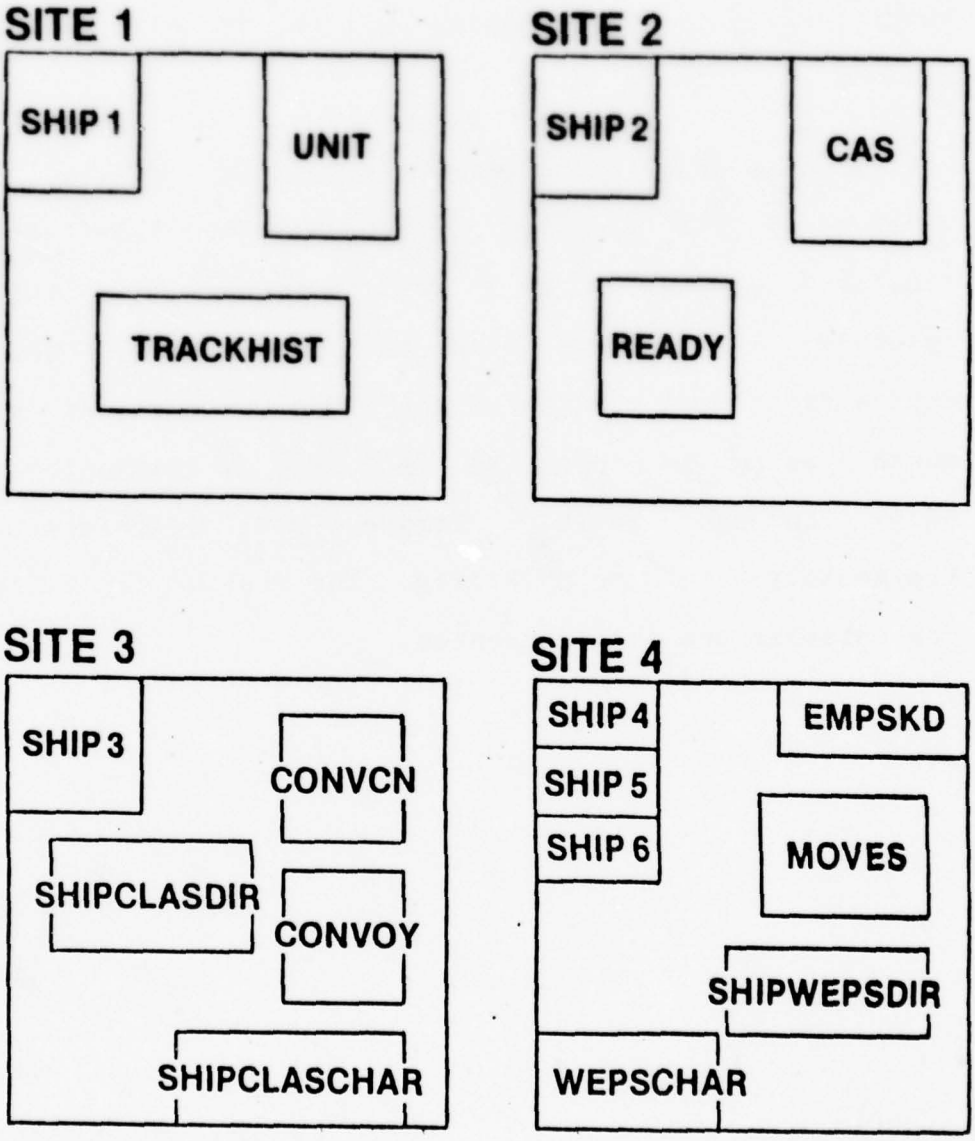
An SDD-1 demonstration runs on five computers on the Arpanet; four of the sites constitute SDD-1 itself and one runs LADDER, the user interface program. Figure 2.5 illustrates the computer and terminal configuration. ISI-E and SRI-KL are TOPS-20 systems; ISI-A and ISI-C are TENEX systems. ISI-E usually runs the TM; all four sites run DMs.

The database used in the demonstration is a distributed version of the Bluefile ([NELC]). The Bluefile is a sanitized extract from a real command and control database. Figure 2.6 illustrates the data distribution chosen for this configuration. One interesting point worth noting is that the SHIP file is fragmented into SHIP1 through SHIP6. Figure 2.7 describes the fragmentation of the SHIP file. The rest of the files in the database are not fragmented.

Data Distribution

Figure 2.6

DISTRIBUTION OF DATA



Fragmentation of SHIP file

Figure 2.7

Figure 2.8 illustrates the picture displayed on the graphics terminal prior to the start of the demo. The figure is self-explanatory in that each large box represents a site and the smaller boxes represent the DMs and TMs at the different sites.

As an example of the demonstration behavior, consider the following query: "Which US ships report casualties involving sonar?" This query should access the READINESS file at site 2 to find ships with REASN equal to 'S' and correlate those ship identifiers (UICVCNs) with US ships in SHIP1. Figure 2.9 illustrates this query. The remainder of this section describes the steps in processing the query.

1. The query is entered at the query entry terminal in English. The query is translated by LADDER ([SACERDOTI]) into Datalanguage and delivered to SDD-1 through an Arpanet connection. All queries enter the system at site 4.

2. The SDD-1 transaction module (TM) produces a relational query from the Datalanguage. This relational query is displayed in the text area of the screen in the color of the TM receiving the query (red in this case).

Initial Display
Figure 2.8

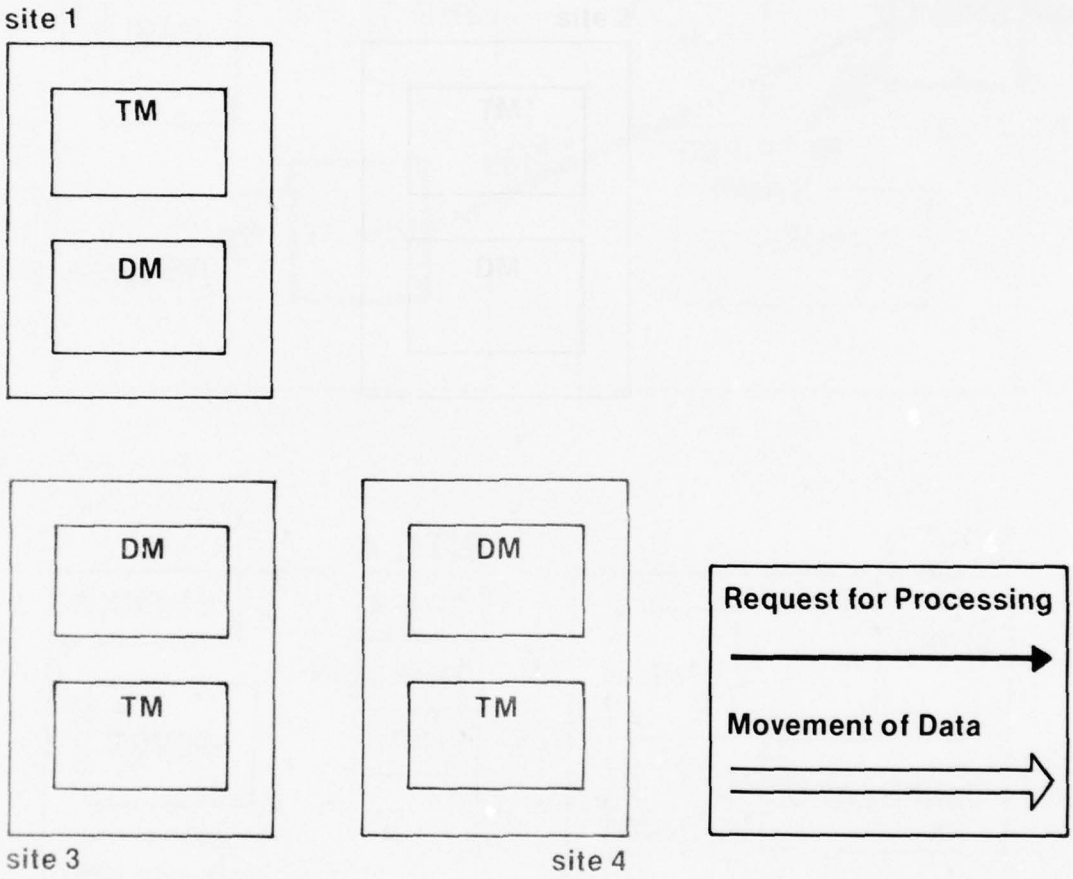
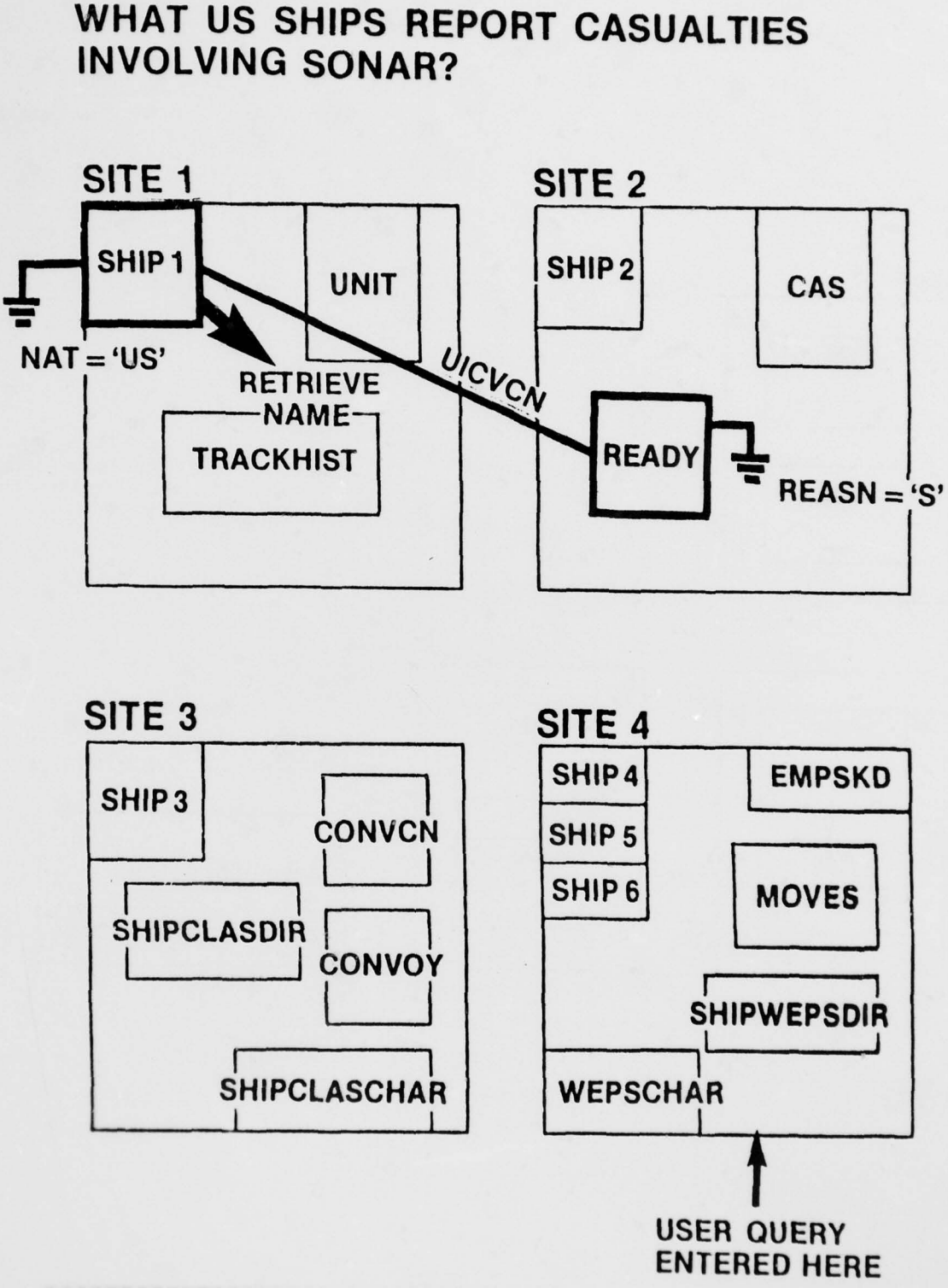


Illustration of Example Query

Figure 2.9



At the same time, the active TM fills with its color and a "Request" arrow is drawn (figure 2.10). The TM produces an access plan that it will use to direct the execution of the query. Before carrying out the execution of the access plan, the system waits for a go ahead signal from the user. The user types any character on the graphics terminal to indicate that the system may proceed.

3. When the go ahead is received, the TM sends out its first set of local processing and move requests. The requests are indicated by the thin arrows. This causes DMs which are executing requests to fill with their color. A data arrow is drawn from the sending DM to its receiving DM. The receiving DM also fills with its color. The processing requests which are sent to each DM are also displayed in the text area in the DM's color. In the example query, the local processing requests records from the READINESS file with REASN equal to 'S' and indicates that the results should be moved to the site of SHIP1. Observe that the access planner realizes that of all the SHIP fragments, only SHIP1 is useful when the qualifier "SHIP='US'" is used. Figure 2.11 shows this stage of the demonstration. The system once again waits for a user go ahead.

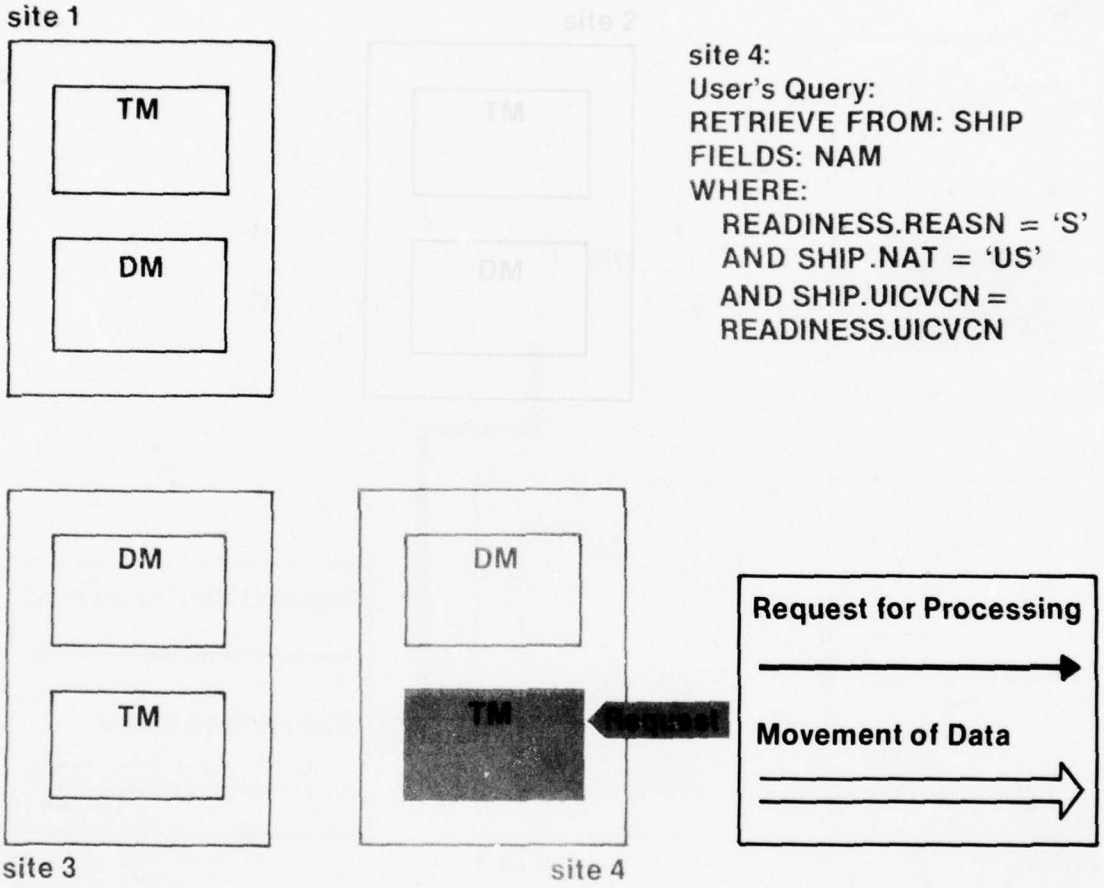
4. As the DMs finish their data moves, they return to their original unfilled state so that when an entire set of moves is complete, only the controlling TM remains filled. Step 3 is repeated continually until all relevant data is at the final site. In our example, only one preliminary move is executed.

5. Having moved all the data to one site, the system sends the user's original query (modified to refer to the temporary files at the final site) to the final site and displays it in the text area (figure 2.12). In our example, the reference to READINESS has been modified to refer to T001, the result of restricting READINESS.

6. Finally, when the DM has the results, they are sent to the controlling TM and back to the user. The results are also reported on the display (figure 2.13).

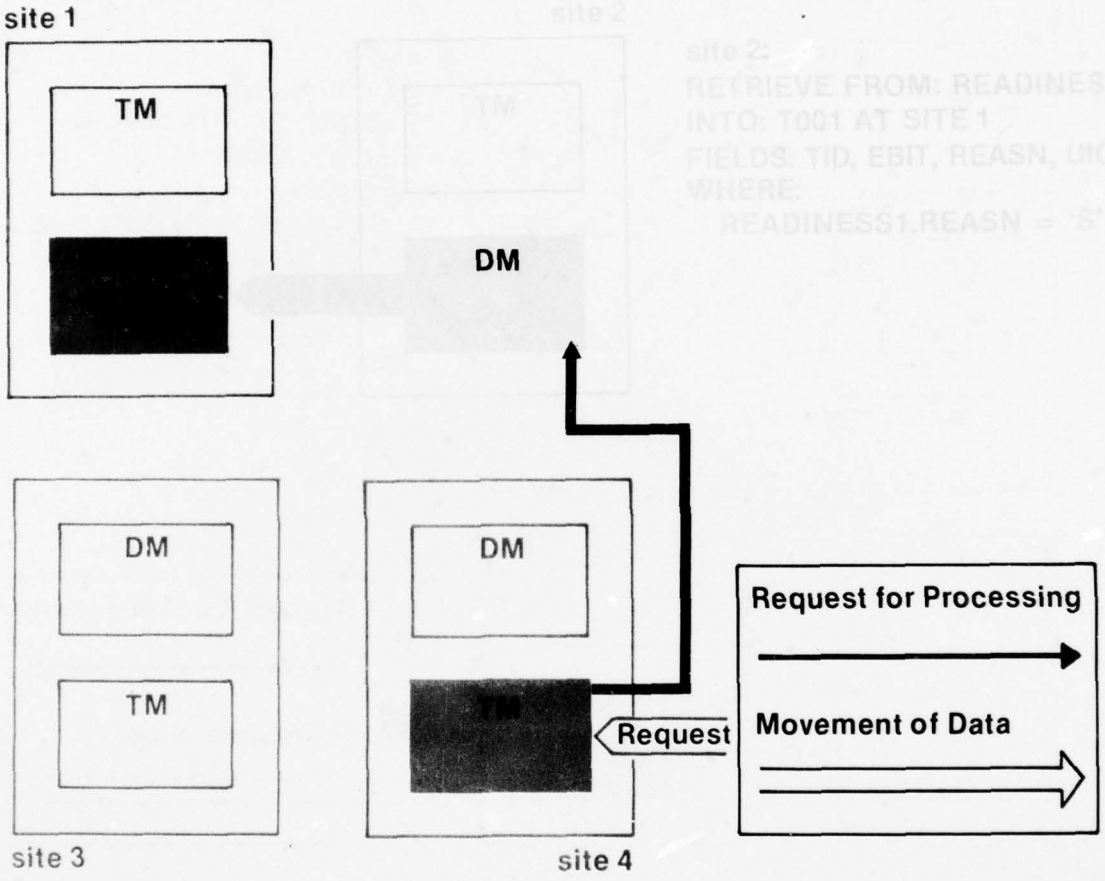
The TM is now ready to receive additional queries.

Query Entered Into System
Figure 2.10

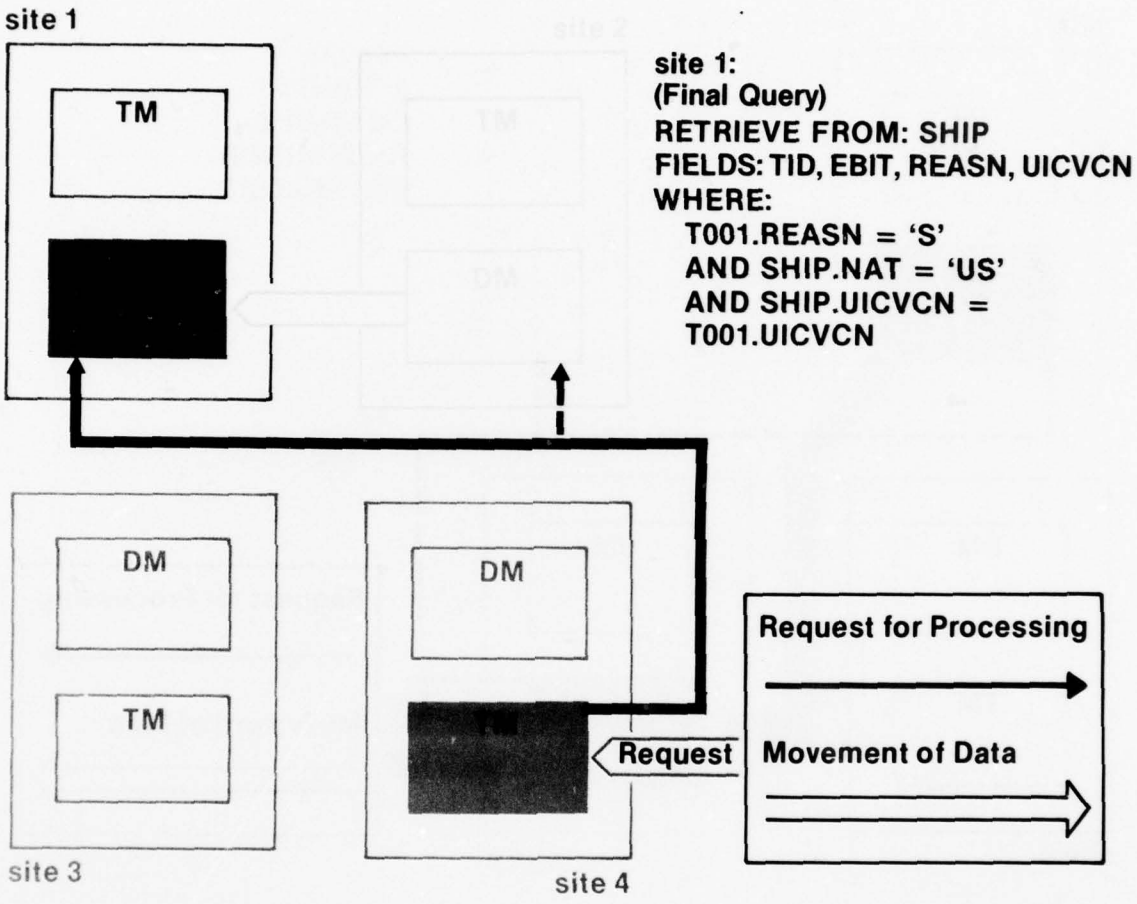


First Execution Step

Figure 2.11

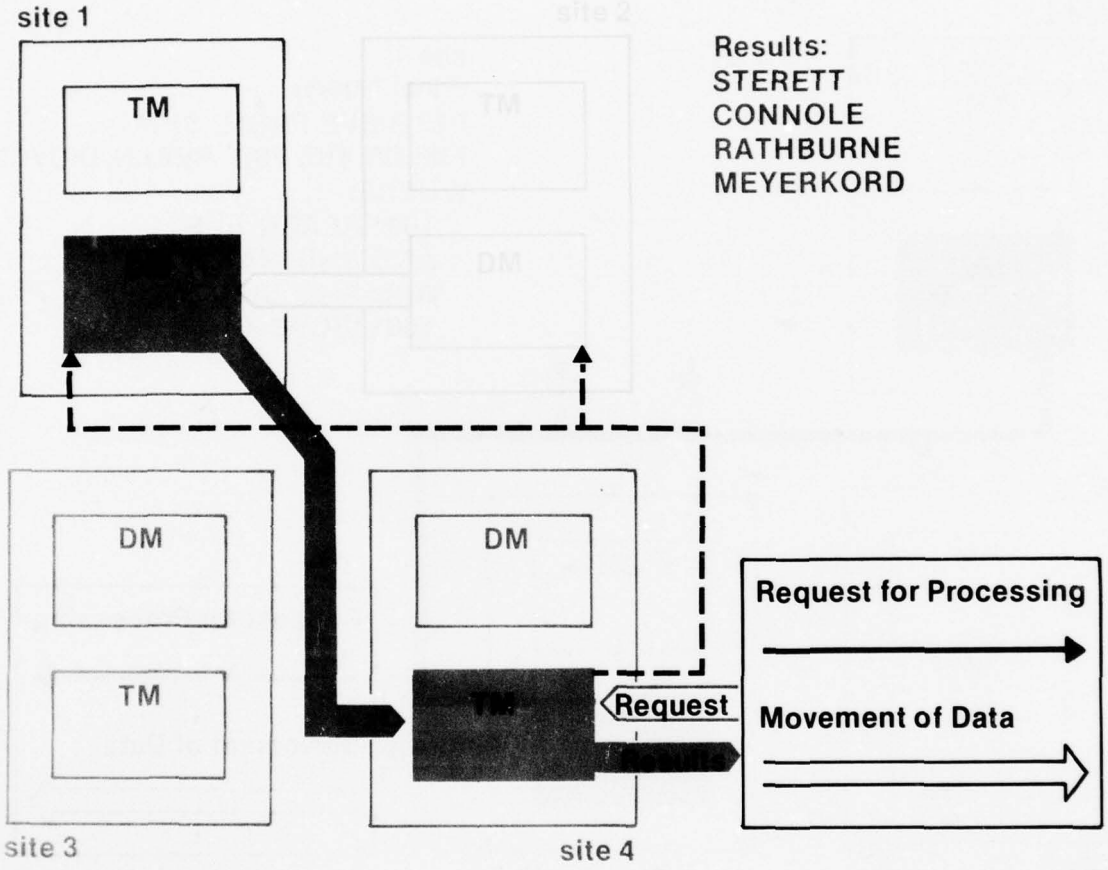


Final Query Execution
Figure 2.12



Results Returned to User

Figure 2.13

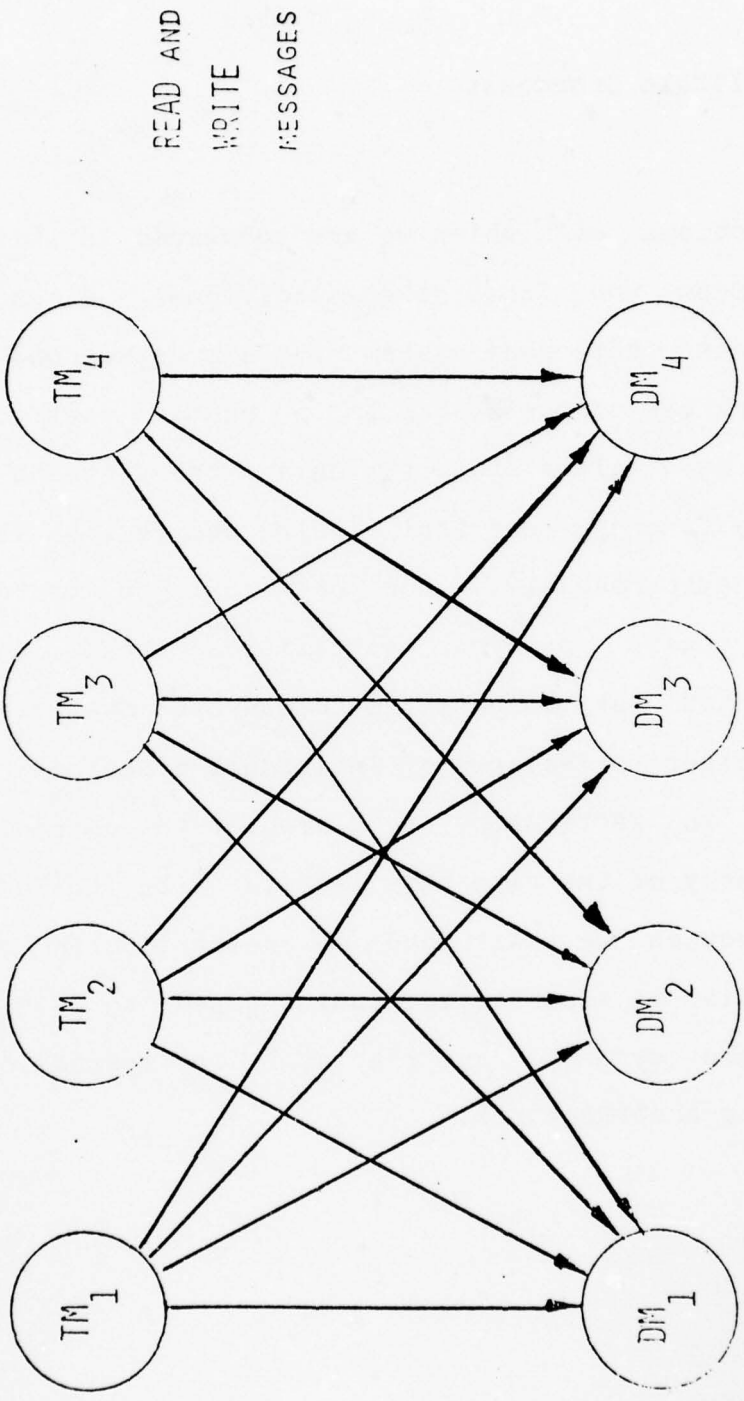


2.4 Reliable Broadcast

The problems with which we are concerned in this section arise from the facts that individual sites in a distributed data base system may fail asynchronously with the operation of the system as a whole; that they may remain in a failed state for an arbitrary amount of time, possibly forever; and that their recoveries will also occur asynchronously. Our principal goal is to provide certain basic system capabilities that enable a distributed data base system to operate in a fashion that is resilient to failures of individual components of the system; in particular, to guarantee the correctness and consistency of the data base despite site failures. In this section, we shall focus on one particular issue that is peculiar to distributed (as opposed to single-site) data base systems, and that afflicts systems of widely differing architectures.

SDD-1 DMs and TMs

Figure 2.14



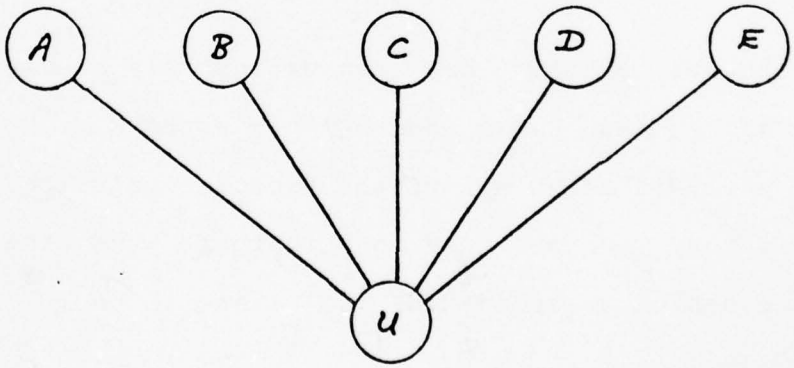
SDD-1 Architectural Overview

The issue we face is that of the reliable broadcast of update messages. Consider the situation of Figure 2.14, where both DM_1 and DM_2 are up. Suppose that TM_1 runs a transaction that changes x at DM_1 from 0 to 1 and y at DM_2 from 0 to 1. Since TM_1 is a sequential machine, it will send out the update messages to DM_1 and DM_2 in some serial order; suppose it first sends to DM_1. Now suppose that TM_1 fails after it has sent the update to DM_1, but before it sends it out the update to DM_2. Then when the transaction at TM_2 accesses x and y, it will find them inconsistent: one will show the effects of transaction TM_1, while the other will not. What is required here is a broadcast facility that enables a package of messages to different sites to be treated as a unit. In particular, the system must guarantee that all the update messages associated with a transaction are sent before the TM fails, or that none are.

The broadcast problem has been extensively studied in a variety of guises by a number of authors. However, careful analysis shows all of the proposed solutions to be unsatisfactory in one way or another, and the basic principle behind a successful solution is complex and subtle.

In presenting our solution to this problem, we wish to explore the issues involved in resolving it and to illustrate the kind of analysis needed to determine the adequacy of proposed reliability schemes in a distributed environment. Therefore, we shall present a series of possible solutions to this problem (some of which have been proposed by other authors), and examine them for their defects. We hope that this process will help develop in the reader an intuition for the strengths and weaknesses of various multi-site reliability mechanisms.

Figure 2.15

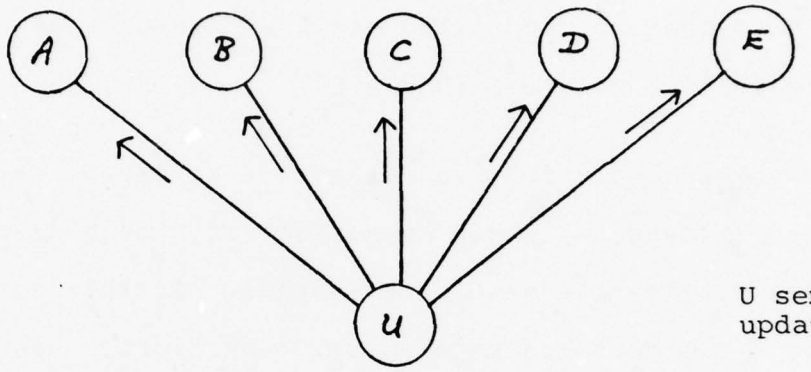


The context for our discussion is shown in Figure 2.15. The transaction module U has completed a transaction and wishes to send update messages to DM sites A,B,C,D, and E. The problem is that U may fail after having delivered updates to some of the sites (e.g., A,B,C) but before sending them to the rest (D and E).

One possible solution is based on an extension of the notion of two-phase commit, a reliability mechanism used in some single-site data base systems. In this scheme, U will send two messages to each DM. The first, the update message, identifies the variable(s) to be updated and their new value(s). Upon receiving an update message, a DM stores it on "stable storage", i.e., in a location which can be expected to survive a system crash. However, it does not yet perform the update; instead, it simply sends an acknowledgement of receipt to U. Once U has received acknowledgements of receipt from all the DM's, it sends a commit message to each one. The commit message signifies that the transaction has been completed and that the update is to take effect. Therefore, upon receipt of the commit message, a DM finally performs the update message it had previously received and stored. Thus, if U fails before sending out all the updates, it will not have sent any commits, and so none of the updates will be performed. The basic operation of this scheme is illustrated in Figures 2.16-2.18.

Step 1: send updates

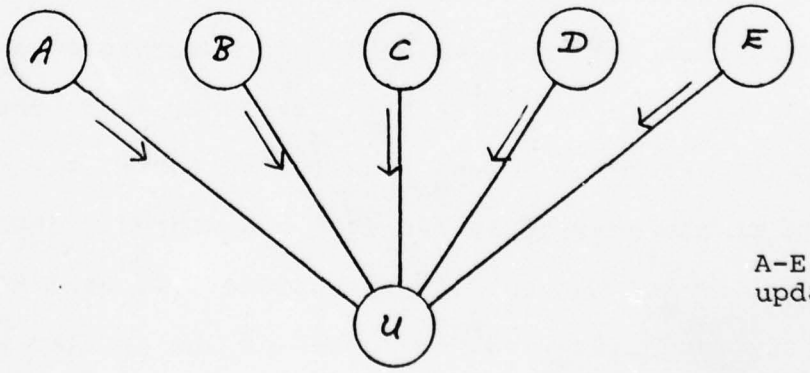
Figure 2.16



U sends updates

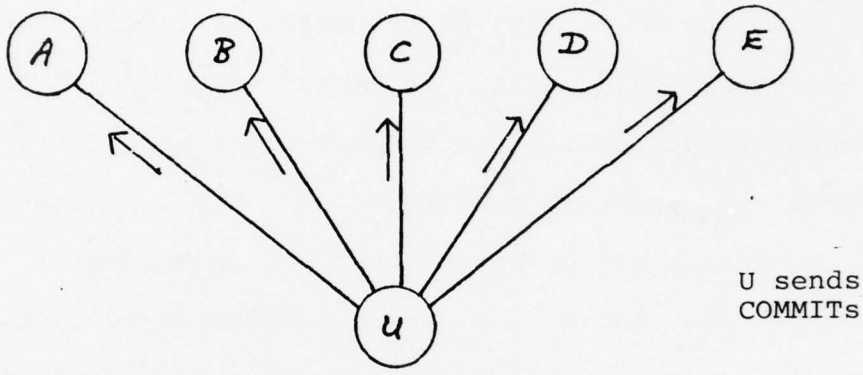
Step 2: updates acknowledged

Figure 2.17



A-E acknowledge updates

Step 3: send commits
Figure 2.18



The terminology commit point is often used in discussing such schemes. This represents a point at which the transaction has been officially completed and its results seen in the system as a whole. If the transaction module fails before it reaches the commit point, then the transaction is aborted; failure of the TM after the commit point should have no effect, since by then the transaction is already assumed to have completed. In the current scheme, the commit point occurs when the TM sends out the first commit message.

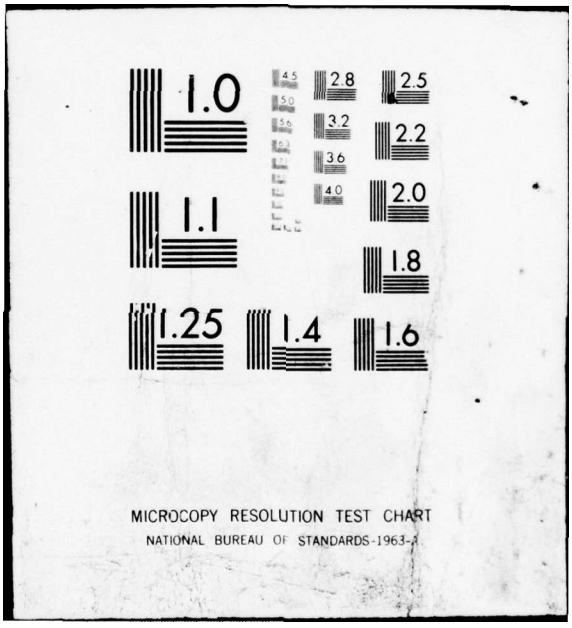
Superficially, this scheme does not seem to represent any improvement over simply sending updates to each DM, which

AD-A060 441 COMPUTER CORP OF AMERICA CAMBRIDGE MASS F/G 15/7
A DISTRIBUTED DATABASE MANAGEMENT SYSTEM FOR COMMAND AND CONTRO--ETC(U)
JUN 78 N00039-77-C-0074

UNCLASSIFIED CCA-78-10 NL

2 OF 2
AD
AO 60441

END
DATE
FILMED
1-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

are installed upon receipt. The troublesome case occurs if U fails after sending out some, but not all of the commits. Then some sites will perform the update, and others will not, resulting in an inconsistency. In other words, if the DM C receives an update but does not receive a corresponding commit, it does not know if U failed before sending out all the updates (in which case C should discard the update it received), or if U succeeded in passing the commit point and sent out some commits, though not the one to C (in which case C should install the update). However, this situation can be resolved; its resolution is based on a critical distinction between update and commit messages. A DM has no knowledge as to whether or not it will receive an update message from a TM; however, once it has received an update, it expects an associated commit from the same TM, and it can take appropriate action if it does not receive one within a reasonable period of time.

The action that a DM A will take in order to resolve a pending update (one for which it has not yet received a commit) will be to inquire of the other DM's involved in the transaction whether or not they have yet received the commit message from U. If any DM responds affirmatively, then A can be certain that U passed the commit point and failed while sending out commits; therefore A can install

the pending update. However, if no DM has yet received a commit, then A can conclude that U failed before passing the commit point, and so A should discard the pending update message.

A DM will initiate such inquiry action to resolve a pending update if it believes that the TM has failed (in which case there is no point in its waiting for the commit message to arrive.) A DM can reach this decision in different ways. The simplest would be based on a time-out mechanism. i.e., after receiving an update message, a DM A would wait for a fixed amount of time, which should be adequate for the TM to finish sending out the remaining updates and the first commits. If A has not received its commit message by the end of this period, it can surmise that the TM failed somewhere between sending A the update and sending it the commit. (This scheme can be modified so that the inquiry action is taken only when some other transaction attempts to read a variable affected by the pending update. At this point, it is necessary to determine whether or not the update is to be installed; until then, however, there is no essential need to resolve the issue.)

Another approach would be for the DM to initiate inquiry about a pending update only when it is certain that the TM

has failed. In many networks, site X can determine whether or not site Y is up by sending it a dummy message; the lack of a reply within a given period indicates that Y has failed. Following this approach, a DM, after receiving an update from U, would enter a state in which it periodically sends probe messages to U. The DM will leave this state either by receiving the commit message or by learning that U has failed. In the latter case, it will institute the inquiry procedure described above.

Observe that this scheme assumes that A knows the identities of the other DM's who were to receive commit messages from U. This information can be supplied to A in the form of an "address list" associated with the update message, indicating all the DM's to which the transaction is sending update messages.

The foregoing presents the basic structure of a two-phase commit scheme. A full specification would have to address such issues as DM's that fail after receiving an update and recover at a later time. But even without considering such complications, it is possible to identify a flaw in this scheme. Suppose that U sends updates to all sites, sends a single commit message to A, and then fails; suppose further that A fails immediately after receiving and acting upon the commit message. (At the time of this

failure, then, A will have the new values of any updated variables.) According to the two-phase commit scheme, site B will eventually decide to resolve its pending update and will issue inquiries to the other DM's involved in the transaction; however, none will respond affirmatively to B's inquiry, since the only site to have received the commit message (namely, A) is down. Consequently, B will discard the update and will continue to utilize the old values of the update variables. (The same holds true for C,D, and E, of course.) However, if A eventually recovers, A will have the new values of the variables, while B (and the others) will have the old ones. This is precisely the kind of inconsistency we are trying to avoid. (It is unsatisfactory to require that B wait until A has recovered in order to finish its polling of the DM's, since A may never be restored to service.)

Because of this case, we say that the two-phase commit mechanism is susceptible to 2-site failure; i.e. the failure of two specific sites (U and A) within a given time window (from the time U sends the commit to A until the time B inquires of A whether it has received the commit) will cause the scheme to fail. The likelihood of the occurrence of such a two-site failure depends, of course, on the failure probability of an individual site,

the transmission times for messages in the network, the length of the chosen time-period, as well as on such considerations as whether or not site failures are independent. In many applications, the likelihood of such a two-site failure may be sufficiently low to make a two-phase commit scheme acceptable. However, we have not been willing to accept schemes that are susceptible to two-site failure. First of all, some of the applications for which the SDD-1 system is intended have characteristics (very high reliability requirements and anticipated failure frequencies) that make the reliability of a two-phase scheme unacceptable. Second, from a conceptual point of view, we are dissatisfied with the notion of there being something "magical" about the number two; we want to better understand the nature of distributed reliability and develop a general scheme that can tolerate an arbitrary number of site failures.

There are two possible ad-hoc patches to the two-phase mechanism, but neither is satisfactory. The first is for B, upon attempting to inquire of A as to the status of the commit and observing that A has failed, to suspend its decision about the status of the commit until A has recovered. This will ensure consistency. However, it will also have the result that any transaction seeking to read an update variable from site B will be unable to do

so until A has recovered. Since we allow for the possibility that A may never recover, this will result in the indefinite delay of some transactions that read variables only from sites that are up; such an anomalous situation is unacceptable. An alternative approach would be to mandate a complex recovery procedure for a data module, which might cause it to discard and undo a commit message that it received before failing. However, the following sequence of events at the DM A causes problems for such a scheme: A receives the commit message for some transaction; A then responds to a retrieval request initiated by some other transaction, and returns the new value of some update variables changed by the previous transaction; A then fails. Then upon A's recovery, it is too late to undo the update; the cow is out of the barn, as the impact of the update has already been felt by the second transaction. To unroll that transaction might lead to cascaded back-ups, the kinds of global rollback that we have explicitly excluded.

In order to address the difficulties raised by the two-phase scheme, we have developed an alternative, 3-phase commit procedure. Its basic principle is as follows. After sending out all the updates, the transaction module U sends out some number of "pre-commit messages", which may go either to DM sites that have been

sent updates, or to other arbitrary sites in the system. The purpose of the pre-commit message is simply to signal the fact that U has passed its commit point (i.e., that it has sent out all the update messages); the receipt of a pre-commit message does not immediately result in any action being taken. After sending out the pre-commits, U sends out commit messages to the DM's, as in the 2-phase scheme.

A DM follows the same procedure that it did under the 2-phase mechanism. After receiving an update, it remains alert to the possibility of the TM's failure. Should the DM decide that the TM has failed then it will have to resolve the pending update it is holding. In the 3-phase scheme, it does so by inquiring not only of the other DM's on the address list, but also of the sites to which pre-commits were to have been sent. Of the former, it inquires whether any have received the commit message; of the latter, whether any have received the pre-commit. In either case, the real goal is to determine whether or not U has passed the commit point. If any queried site responds affirmatively to the question asked it, then the inquiring DM installs its pending update; otherwise, the DM discards it.

Observe that this scheme is not susceptible to two-site failure in the way the 2-phase scheme was. There obviously can be no opportunity for error if U has not passed the commit point, because then no site will ever decide to install the update. If U has passed the commit point, an inconsistency can result only if U, all the sites that have received pre-commit messages, and some DM that has installed its update, all fail within a given time-frame. Clearly, this calls for the failure of more than two sites. In all other situations, the DM's will have a mutually consistent view of the update.

However, inconsistencies can be brought about by the failure of 3 sites in a small time window. Consider the following scenario. The TM U sends out all the updates and a pre-commit to site P, and then U fails; DM site A then decides to resolve its pending update, and learns from P that U has passed the commit point; then P and A both fail before any other DM has the opportunity to learn from either of them that U has passed the commit point. Then the (failed) DM site A will have the new values of its update variables, while all other DM's will have the old ones.

The essential problem with the scheme just presented is that the site A was permitted to install its update upon

learning from P that U passed the commit point, before any other DM sites had the opportunity to learn the same fact. That is, there was a point in time at which only two sites A and P knew that U passed the commit point, and one of them (A) was a DM that acted upon its knowledge. Simultaneous failure of these two sites prevents others from learning that U has passed the commit point and so causes inconsistencies. Our approach to rectifying this problem is based on distinguishing between a DM's knowing that U passed the commit point and its acting on this knowledge (by installing a pending update). At times, we shall separate knowledge from action in time, by requiring that a DM wait for a period of time of length d before installing a pending update after it has learned that U passed the commit point. This waiting period is not required if the DM A acquires its knowledge from U directly (via a commit message), but only if A learns the facts "secondhand" (e.g., from a site that received a pre-commit message). In this case, the purpose of A's wait is to give the other DM's time to learn the facts before A acts on its knowledge; A wants to avoid acting on information that it alone possesses, lest it subsequently fail, leading to an inconsistency.

The parameter d, which specifies how long a DM is to wait, should be selected as follows. It ought to be long enough

to allow all the other DM's to decide (by time-out or probing) that U has failed, for them to inquire of other sites if U sent any pre-commits or commits before failing, and to receive an affirmative response from A to this inquiry. At this point, A can safely install its pending update, for even should it fail, the other DM's should know that U has passed the commit point.

Numerous issues in this mechanism remain to be examined. However, in order to consolidate the ideas that have been introduced so far, we present the following specifications. First, we describe the operation of the TM U in completing a transaction; these actions are to be taken after the TM has computed the new values of the update variables and is prepared to send them to the appropriate DM's.

1. U sets its state flag to "updating"; this is maintained on "stable storage."

2. U sends the appropriate update message to each relevant DM, including with it the list of all other DM's to which updates are being sent and the list of all sites to which it plans to send pre-commit messages.

3. U waits for an acknowledgement of receipt (and installation on stable storage) of each update. If a DM is down, U "spools" its update message by causing it to be stored at a number of sites in the system. (See [HAMMER and SHIPMAN] for a discussion of spooling.)

4. U sets its state flag to "committing"; U then issues pre-commit messages to certain sites in the system; the number of such messages is a parameter that affects the reliability of the system as a whole. (See discussion below.)

5. U waits for acknowledgements that all the pre-commits have been received.

6. U issues commit messages to all the DM's. After receiving an acknowledgement of receipt from some DM, U sets its flag to "acknowledged".

This completes the description of U's operation. The principles of operation of each DM A, in processing updates, follows.

1. A receives an update message from U, (which includes an address list) saves it on stable storage, and sends an acknowledgement to U.

2. A enters a state in which it is sensitive to U's failure. A leaves this state either by receiving a commit message from U (in which case it installs the update, acknowledges U, and terminates) or by deciding that U has failed. The latter may be accomplished either by monitoring U or by a time-out. The remaining steps specify A's behavior if it has decided that U has failed.

3. A issues inquiries to all other sites, on the address list, asking if any have received a pre-commit message or a commit message from U that is associated with the pending update. A will not wait for a reply from a down site, (one that does not reply within a given time period.)

4. There are three possible outcomes of this inquiry.

   a. If all sites respond negatively, then A discards the update.
   b. If some site responds that it has received a commit, then A immediately installs the update. If later queried about the update, A will reply that it received the commit, for the update

c. If some site responds that it has received a pre-commit, but no site responds that it received a commit, then A will wait for a period d and then install the update. If queried about the update within this interval, A will respond that it has received a pre-commit; if queried after the interval, A will reply that it has received a commit.

5. After installing the update, A sends an acknowledgement to U.

This completes our preliminary specifications. Several essential questions remain to be addressed; the first of these is how U and A are to recover from failure. The issues are these.

Suppose that A received the update but failed before resolving it (i.e., before either receiving the commit or receiving enough information from other sites to enable it to install or discard the update). How should A proceed when it recovers? Similarly, suppose U fails after it passes the commit point? What should U do upon recovery? (Clearly, if U fails before passing the commit point, then on recovery it simply aborts the transaction.) The issue

in the first case is how A, upon recovery, is to determine the status of the pending update; in the latter, it is whether U, upon its recovery, has to take some steps to assure consistency among the DM sites.

Suppose that A recovers with a pending (unresolved) update; that is, after recovering and processing all messages that were sent to it while it was down, A observes that it still has an update message for which it has not yet received a commit. This situation can occur in two ways. The first would involve the update message being sent to A while A was down; the second would involve A failing after receiving the update but before resolving it. In either event, U must have failed sometime after sending A its update but before sending its commit.

The obvious step for A to take upon its recovery is to send inquiries to the sites on the address list of the update. By the time A recovers, these other sites should have resolved the update and can inform A of its resolution. But there are two points that must be accounted for. The first is that the other sites may no longer remember what became of the update about which A is inquiring. This might occur if U had recovered and successfully completed some other transactions before A recovered; then A's inquiry about its pending update may

refer to very old information. Of course, each DM could maintain a complete history of all the updates it has received, but that would be very expensive. We would like each DM to be forced to maintain only a limited amount of information in order to reply to queries sent by other DM's upon their recoveries. The other problem involves "short address lists." That is, if only a small number of sites were to be recipients of the update message, then upon A's recovery, it may find that they are all down and consequently unable to reply to its inquiry. The resolution of these two issues entails impact on the recovery procedures for the TM U as well.

We shall stipulate that the inquiry message sent by A on its recovery to the other DM's is, "What is the timestamp of the last committed update message that you have received from the TM U?" This imposes a limited storage requirement on each DM: namely, that for each TM, it maintain a buffer of length one, describing the last committed update the DM has received from that TM. Then there are three possibilities regarding the replies that A will get to its inquiry. The last committed message that the responder has received from U may be timestamped before the pending update that A is trying to resolve; this means that update was not committed while A was down, and so A may discard it. (We can avoid the possibility of

A's inquiring in the middle of the resolution process in a number of ways.) If the last committed update has the same timestamp as the pending update, then that update was committed while A was down, and so A should install it. If the last committed update is timestamped later than the pending update, the implication is that U recovered from its failure while A was down, and ran additional transactions. How can A then determine if the pending update was committed? We can make this possible by stipulating that whenever U fails in the process of committing an update, upon its recovery U will send out commit messages to all those DM's to which it did not send commits before it failed. (This will be done prior to running any additional transactions.)

Thus, if A receives as a response to its inquiry, that U has committed an update timestamped later than the one that A has pending, then A can conclude that U recovered at some point while A was still down. If the pending update had indeed been committed, then U ought to have sent commit messages to all DM's that had not yet been sent then (including A). However, A did not receive any such commit (if it had, the update would no longer be pending). Consequently, A can conclude that the update was not committed and so can discard it.

There is one point in this scheme that requires resolution: namely, upon its recovery, how does U know whether it should send out commit messages? The problem is that U may not be sure if the update was committed while it was down. If U failed before sending out any pre-commits, then it can be sure that the update was not committed. If, on the other hand, it has failed after sending out all n pre-commits, then it has passed the commit point and the update is assumed to be committed. However, suppose that U had failed while sending out the pre-commit messages; then the update may or may not have been committed, depending on the pattern of failures among the pre-commit sites. Upon its recovery, U will not immediately know if the update was committed (and so if it should send out commit messages).

We shall have U resolve this uncertainty by itself instituting inquiries to the DM's on the address list, to determine if they committed or discarded the update in question while U was down. This inquiry will also be of the form "What is the timestamp of the latest committed update that you have received from me?" If any site replies with the timestamp of the update that U is trying to resolve, then that update was committed while U was down. Note that we have moved away from a fixed commit point for U before which the update is not committed and

after which it is. We have a gray region, from the sending of the first pre-commit to the sending of the last one, during which the update is in an ambiguous state. If U fails in this period, then on its recovery, U must ask the DM's what conclusion they have reached about its status.

The "short address list" problem surfaces here too. If only a few sites were on the address list of the update, then none may be up to respond to U's inquiry. There are a variety of ways in which this problem might be handled. The approach that we have chosen is to pad short address lists with dummy updates to other sites until a list of acceptable length is achieved. That is, if an update is only to be sent to sites A and B, we would also send it to some additional sites (say C, D,and E); the updates sent to these sites, however, would not cause any variables to be updated. In all other respects, these updates are treated like the others, and so a larger number of DM's become available for inquiring as to the status of the update (either by a recovering DM with a pending update or by a recovering TM that failed while sending pre-commits). The selection of the desired length of the address list should be based on the failure probabilities of the sites in the system and the required level of reliability.

To completely specify the algorithm, there are some other situations that must be addressed. One is, what if one of the pre-commit sites fails? We shall stipulate that on its recovery it discards its pre-commit message; and while it is down, the other sites simply ignore it. Another issue that must be addressed is how this algorithm works if any of the data module sites are down, either at the time that messages are sent to them, or after they receive them. This problem is solved with spoolers which are described in [HAMMER and SHIPMAN].

3. Enhancement of Datamodule 1

This section describes initial work on a project entitled "The Enhancement of Datamodule 1". The focus of this project is to study the Datacomputer's performance in the SDD-1 environment and other command and control applications with the goal of suggesting enhancements that will improve its performance. The study procedure we are using is a formal analytic process aimed at identifying request processing bottlenecks, suggesting techniques to relieve these bottlenecks, and evaluating the cost-effectiveness of the suggested improvement techniques.

3.1 Study Procedure

The study procedure steps being followed are:

1. Model building -- determining a simple picture of the way the Datacomputer performs.

2. Requirements Analysis -- determining the performance requirements of the command and control community.

3. Measurement -- calibrating the model with the performance parameters of the current Datacomputer.

4. Sensitivity Analysis -- determining where the system bottlenecks are using the calibrated model and the requirements analysis data.

5. Enhancement option generation -- proposing candidate actions for improving performance.

6. Enhancement option analysis -- determining the effects on the model of adopting each performance enhancement option.

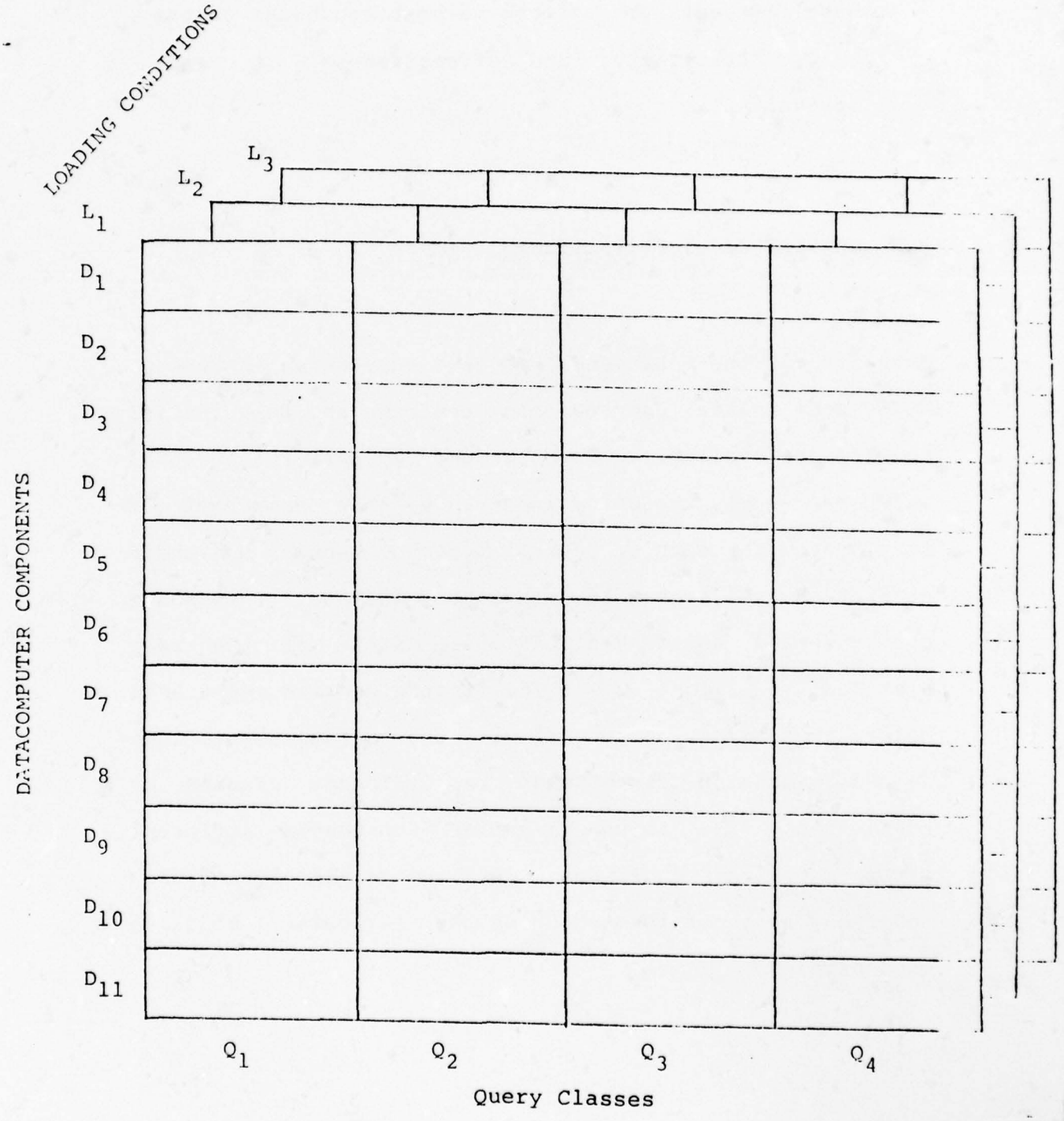
7. Evaluation and recommendation -- recommending a specific set of actions to be taken based on the cost, flexibility, and effectiveness of each option.

3.2 Initial Model Definition

Even though the enhancement project only began in June of 1978, some initial progress has been made and some initial results are available. The Datacomputer performance model which represents the delay incurred by each component of the system for each of several types of queries and under each of several system loads has been defined. This model can be viewed as a 3-dimensional matrix of the sort pictured in figure 3.1. The first dimension represents major processing components of the Datacomputer, the second dimension corresponds to different classes of queries, and the third dimension represents different system loading conditions. An initial version of this model has been constructed using the following units on each axis:

Datacomputer Model

Figure 3.1



1. Datacomputer components -- the system has been divided into twelve major processing components. There are three kinds of components that have been identified:

   - utilities -- Processing actions that are invoked from many different parts of the system and take up a reasonable amount of resources were put in this group. These include printouts, page reads, system overhead etc.

   - modules -- Components in this category are relatively large sections of code that perform a well-defined function such as "compiler" or "inversion processor".

   - subroutine -- If a single subroutine appears to be a major bottleneck, it may be considered a component in the model.

2. Query classes -- The initial version of the model deals with two kinds of queries: queries produced by LADDER ([SACERDOTI]) on the Bluefile ([NELC]) and transactions produced by the WES utility, a war games simulator used at ACCAT. This dimension will

be expanded to include LADDER queries on the ACCAT FC database and typical Datalanguage requests produced by SDD-1.

3. System Loading -- Based on previous observations and interactions with NOSC, we decided to emphasize paging as the important load factor in the model. Three levels of paging were chosen:

   - light -- essentially no paging outside the Datacomputer
   - medium -- a moderate amount of paging produced by one page-bound competing job on an otherwise empty system.
   - heavy -- a large amount of paging produced by two such competing jobs.

   Higher levels of paging were investigated but did not produce dramatically different results than the one we used.

Measurement experiments were conducted on the Datacomputer to populate the initial model. In order to perform these experiments, modifications were made to the Datacomputer to cause it to print out information about CPU utilization, page faults and elapsed real time. These

printouts occur whenever a new processing component is invoked. This information is then used to compute totals of these quantities for each component.

3.3 Initial Measurements

This modified Datacomputer was used to run experiments on the queries produced by LADDER. Extensive experiments were conducted on thirteen such queries (each accessing two files and one port) and a few of the experiments were run on one hundred queries as further verification of the results on the smaller sample. Several characteristics of these queries and the database are important to the analysis of the results:

- The files are short (200 or fewer records per file).
- Each file has many inverted fields.
- All the requests are retrievals.
- All files were open and stayed open during the tests. No OPENs or CLOSEs were executed.

The first set of results show the CPU time used by different parts of the Datacomputer. For expository purposes, the twelve processing components were collapsed into four in the figure 3.2. The services section mentioned in the figure includes all Datacomputer utilities, I/O routines and other system functions. Because services utilized about two-thirds of the time, it was subdivided in figure 3.3.

Division of CPU Utilization Figure 3.2

Component	Percent of CPU time
Services	62
Inversion Proc.	15
Compilation	20
Other	3
TOTAL	100

Services CPU Breakdown Figure 3.3

Component	Percent of CPU time
Page reads	22
Scratch file reads	19
Opening files/ports*	8
Closing files*	4
Listening on network	6
Buffer calls	3

*these refer to internal opening and closing, not the OPEN and CLOSE commands

From figure 3.3, it is obvious that page reads are very expensive. In order to get a better handle on why this is the case, an experiment was conducted comparing the CCA Datacomputer and a standard TENEX Datacomputer (like the one in the ACCAT). Unlike the TENEX Datacomputer, the one at CCA maintains its own disks separate from those on TENEX. Therefore, it was possible to compare the time to do a page read on the two systems to determine how much of that time was TENEX overhead. Figure 3.4 shows the results of this test.

Comparison of TENEX & DC page reads — Figure 3.4

TENEX Page Read

Component	CPU time in ms
Buffer manipulation	3.5
Reading Page	9.0
Other	1.5
TOTAL	14.0

Datacomputer Page Read

Component	CPU time in ms
Buffer manipulation	3.5
Reading Page	2.0
Other	1.5
TOTAL	7.0

Internally opening and closing files also incurs a significant time cost in these experiments. Further

investigation indicated that a major reason for this cost was changing the access mode of Datacomputer directory pages. In general, these pages are kept read-only for reliability reasons. However, whenever they are accessed, the Datacomputer updates the "last read" slot in the page. This process entails making the page writable, writing the data and time and then making it read-only again. Experiments indicated that about 75% of the time opening and closing files was spent in changing the modes of pages. This seems to be an area where reliability could be traded off against performance.

The next set of experiments measured Datacomputer performance at different levels of paging activity. Three levels of paging were established as described earlier. The results are summarized in figures 3.5 and 3.6. Figure 3.6 also indicates clearly which processing components are most sensitive to paging activity.

Paging Activity Variation Figure 3.5

	Light	Moderate	Heavy
CPU(ms)	11441	11669	12703
Real(ms)	20000	49470(2.5)*	80919(4.0)
Page faults	197	363(1.9)	571(2.9)

*numbers in parenthesis indicate the factor of increase over light.

Paging activity on a component basis Figure 3.6

	Light	Moderate	Heavy
Printouts	229	1309(6.0)	2193(9.9)
Context	128	1071(8.0)	1504(12.0)
Precompile	551	3868(6.5)	5628.0(12.0)
Compile	950	5372(5.5)	3697(3.5)
Inst. Gen.	1426	5060(3.5)	7176(5.0)
Overhead	2078	6900(3.3)	10627(5.0)
Inversion	7731	14782(2.0)	19560(2.6)
Page read	4914	11775(2.4)	20541(4.4)
Scratch read	3429	4923(1.4)	7217(2.1)
Read Only pg.	597	526(.9)	526(.9)
Writable pg.	599	928(1.6)	2414(4.1)
Open Buffer	112	1188(10.6)	1165(10.6)

Initial analysis of the measurements made on the LADDER queries indicates that overhead both in TENEX and in the Datacomputer is very significant for queries on the small files used. Further investigations using the ACCAT FC database will establish whether or not this is also the case with large files.

The WES transactions that we examined at had different characteristics than the LADDER queries:

- The files were very small.
- All the requests involved appending or updating to files.

- No inversions were involved at all.
- Each request accessed at most 400 records.
- Each request accessed at most 20 data pages.
- The requests were all precompiled.

Since the WES data had to be produced on the ACCAT Datacomputer, we were not able obtain as much measurement data as we could on the LADDER queries. If mored data is required during the study, a trip will be made to NOSC to perform experiments.

WES operates by running a series of requests every cycle through the simulation. Seven different precompiled requests are run each time. Figure 3.8 lists the requests and what they accomplish; figure 3.9 shows CPU and real time for the compile and run phases of each request. The compile phase in this case includes reading the precompiled request and finishing the compilation. The runtime part includes essentially everything else involved in the request. These results indicate that even the small amount of time required to setup a precompiled request is significant when we are dealing with requests on such small files. Once again overhead is a very important cost factor.

WES Requests Figure 3.7

TRANS	transfer records into temporary file
UPDATE	update position of ships and aircraft if they are already in the database
HIST	add records for new ships and aircraft to trackhist file
CONT	update the contact file
CASR	update the casualty file and the readiness file
POSAPPEND	append to the position file all records not found during the update phase
UNITUPDATE	update the unit file

WES Requests CPU time and real time Figure 3.8

	Compiler CPU / REAL	Runtime CPU / REAL
TRANS	1727 / 2799	2792 / 28898
UPDATE	3400 / 7085	5827 / 8824
HIST	3550 / 6166	4906 / 10669
CONT	2200 / 4150	2620 / 6203
CASR	3550 / 3968	4085 / 8594
POSAPPEND	3650 / 10211	4628 / 8027
UNITUPDATE	3100 / 11032	2530 / 4067

3.4 Future Work

Some possible enhancements associated with cutting overhead and improving paging activity clearly suggest themselves based on our preliminary findings. The next step in this study will be to expand the query classes that we are considering to include the FC database and the SDD-1 environment. This will give us enough data to begin serious consideration of other enhancements.

References

[ASTRAHAN et al]
Astrahan, M.M.; et al. "System R: Relational Approach to Database Management", ACM Transactions on Database Systems, Vol. 1 No. 2, June 1976, pp. 97-137.

[BBN]
Bolt Beranek and Newman, "MSG: The Interprocess Communications Facility for the National Software Works", Report No. 3485., Bolt Beranek and Newman, 50 Moulton Street, Cambridge, Massachusetts 02138.
(Also Available
from Massachusetts Computer Associates, 26 Princess Street, Wakefield Massachusetts 01880, as Document No. CADD 7612-2411).

[BERNSTEIN et al a]
Bernstein, P.A.; Goodman, N; Rothnie, J.B.; and Papadimitriou, C.A. "Analysis of Serializability in SDD-1: A System for Distributed Databases (The Fully Redundant Case)", First International Conference on Computer Software and Applications (COMPSAC 77), IEEE Computer Society, Chicago Illinois, November 1977.
(Also available
from Computer Corporation of America, 575 Technology Square, Cambridge Massachusetts 02139 as Technical Report No. CCA-77-05.)

[BERNSTEIN et al b]
Bernstein, P.A.; Rothnie, J.B.; Shipman, D.W.; and Goodman, N., The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The General Case), Technical Report No. CCA-77-09, Computer Corporation of America, 575 Technology Square, Cambridge Massachusetts 02139.

[CCA a]
Computer Corporation of America, A Distributed Database Management System for Command and Control Applications: Semi-Annual Technical Report, Technical Report No. CCA-77-06, Computer Corporation of America, 575 Technology Square, Cambridge, Massachusetts 02139.

[CCA b]
Computer Corporation of America, A Distributed Database Management System for Command and Control Applications: Semi-Annual Technical Report 2, Technical Report No. CCA-78-03, Computer Corporation of America, 575 Technology Square, Cambridge, Massachusetts 02139.

[CODD]
Codd, E.F., "A Relational Model of Data for Large Shared Data Banks", CACM, 13 (1970), pp. 377-387.

[ESWARAN et al]
Eswaran, K.P.; Gray, J.N.; Lorie, R.A.; Traiger, I.L. "The Notions of Consistency and Predicate Locks in a Database System", Communications of the ACM, Vol. 19, No. 11, November 1976.

[GRAY et al]
Gray, J.N.; Lorie, R.A.; Putzolu, G.R.; Traiger, I.L. "Granularity of Locks and Degrees of Consistency in a Shared Database", Report from IBM Research Laboratory, San Jose California, 1975.

[HAMMER and SHIPMAN]
Hammer, M.M.; and Shipman, D.W., "Resiliency Mechanisms in SDD-1", Technical Report in progress. Computer Corporation of America, 575 Technology Square, Cambridge Massachusetts 02139.

[HEWITT]
Hewitt, C.E. "Protection and Synchronization in Actor Systems", Artificial Intelligence Laboratory Working Paper No. 83, Massachusetts Institute of Technology, November 1974.

[MANNA]
Manna, Z. Mathematical Theory of Computation, McGraw-Hill, New York, 1974.

[MARILL and STERN]
Marill, T.; and Stern, D. H., "The Datacomputer: A Network Data Utility", Proceedings AFIPS National Computer Conference, AFIPS Press, Vol. 44, 1975.

[NELC]
NELC, "A Relational Model for an st Sea Commander's Tactical Database", Project Scientist: Garrison Brown, SEI, October 29, 1976.

[PAPADIMITRIOU et al]
Papadimitriou, C.A.; Bernstein, P.A.; and Rothnie, J.B., "Some Computational Problems Related to Database Concurrency Control", Conference on Theoretical Computer Science, University of Waterloo, Waterloo Ontario, August 1977.

[ROTHNIE et al]
Rothnie, J.B.; Goodman, N.; and Bernstein, P.A. "The Redundant Update Algorithm of SDD-1: A System for Distributed Databases (The Fully Redundant Case)", First International Conference on Computer Software and Applications (COMPSAC 77), IEEE Computer Society, Chicago Illinois, November 1977.
(Also available
from Computer Corporation of America, 575 Technology Square, Cambridge Massachusetts 02139, as Technical Report No. CCA-77-02).

[ROTHNIE and GOODMAN]
Rothnie, J.B.; and Goodman, N. "An Overview of the Preliminary Design of SDD-1: A System for Distributed Databases", 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, University of California, Berkeley California, May 1977.
(Also available
from Computer Corporation of America, 575 Technology Square, Cambridge Massachusetts 02139, as Technical Report No. CCA-77-04.)

[SACERDOTI]
Sacerdoti, E. D. "Language Access to Distributed Data with Error Recovery," Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts, August 1977.

[STONEBRAKER et al]
Stonebraker, M.; Wong, E.; Kreps, B.; and Held, G. "The Design and Implementation of INGRES", ACM Transactions on Database Systems, Vol. 1, No. 3, September 1976, pp. 189-272.

[WONG] Wong, E. "Retrieving Dispersed Data from SDD-1: A System for Distributed Databases", 1977 Berkeley Workshop on Distributed Data Management and Computer Networks, Lawrence Berkeley Laboratory, University of California, Berkeley California, May 1977.

(Also available
from Computer Corporation of America, 575
Technology Square, Cambridge Massachusetts 02139,
as Technical Report No. CCA-77-03.)